Vergleich von Time Series Databases und Event Stores

Masterarbeit zur Erlangung des Master-Grades
Master of Science im Verbundstudiengang Wirtschaftsinformatik
an der Fakultät für Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln

vorgelegt von:	Kathrin Schinker
Matrikel-Nr.:	11118799
E-Mail:	kathrin@schinker-online.de

eingereicht bei:	Prof. Dr. Heide Faeskorn-Woyke
Zweitgutachter:	Jan Strohschein

Köln, den	08.08.2019

Technology
Arts Sciences
TH Köln

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Köln, 08.08.2019 

Ort, Datum

Rechtsverbindliche Unterschrift

Abstract

Aufgrund ihrer aktuellen Bedeutung im Zusammenhang des Internet of Things werden in der vorliegenden Arbeit Time Series Databases und Event Stores miteinander verglichen. Ziel ist, die Gemeinsamkeiten und Unterschiede der beiden Arten von Datenbank Management Systemen herauszustellen.

Der erste, theoretische Teil des Vergleichs erfolgt anhand der funktionalen Kriterien Speichersystem, Performance und Funktionen sowie der nicht-funktionalen Kriterien Usability und Support. Im zweiten Teil des Vergleichs wird anhand eines konkreten Anwendungsfalls untersucht, ob sich Time Series Databases und Event Stores gleichermaßen für die Speicherung und in einem zweiten Schritt für die Abfrage von Zeitreihendaten eignen.

Zumal der theoretische Vergleich Unterschiede zwischen einzelnen Time Series Databases und Event Stores in Bezug auf die betrachteten Kriterien erkennen lässt, wird für den praktischen Vergleich unter Berücksichtigung der im konkreten Anwendungsfall gegebenen Anforderungen nur die am besten geeignetste Time Series Database (*InfluxDB*) und der am besten geeignetste Event Store (*Event Store*) ausgewählt. Der praktische Vergleich zeigt, dass die Zeitreihendaten im konkreten Anwendungsfall zwar in beiden Arten von Datenbank Management Systemen gespeichert werden können, die Nutzung der auf Zeitreihendaten spezialisierten Time Series Database *InfluxDB* jedoch offensichtliche Vorteile gegenüber dem *Event Store* aufweist.

Inhalt

Tabellenverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

API	Application Programming Interface
BLOB	Binary Large Object
CDA	Clinical Document Architecture
CLI	Command Line Interface
CQRS	Command Query Responsibility Segregation
DBMS	Datenbank Management System (Database Management System)
GB	Gigabyte
GUI	Graphical User Interface
HDD	Hard Disk Drive
InfluxQL	Influx Query Language
IoT	Internet of Things
KB	Kilobyte
MB	Megabyte
RDBMS	Relationales Datenbank Management System
SQL	Structured Query Language
SSD	Solid State Drive
TSDB	Time Series Database
UI	User Interface
UUID	Universally Unique Identifier

1 Einleitung

1.1 Motivation

In der Ära von Big Data sind eine wachsende Anzahl unterschiedlicher Datenquellen und folglich auch stetig steigende Datenmengen keine Neuheiten mehr. Insbesondere Sensoren, welche zeitbezogene Daten in Echtzeit generieren und erfassen, werden nicht nur für die Industrie 4.0, sondern für jegliche Anwendung des Internets der Dinge (Internet of Things (IoT)) immer relevanter. Damit steigt auch die Bedeutung der Datenbank Management Systeme (DBMS), mit denen die im Zeitverlauf generierten Daten, welche somit Zeitreihen darstellen, gespeichert und verarbeitet werden können.[1] In diesem Zusammenhang sind Time Series Databases (TSDBs) zu nennen. Diese Art von DBMS ist auf die Speicherung und Abfrage von Zeitreihendaten spezialisiert, was sich durch das Vorhandensein von primär auf Zeitreihen ausgerichteten Funktionen, bspw. für die Verdichtung bzw. Aggregation von Zeitreihendaten, zeigt.[2] Da die mit den Sensoren erfassten Daten zudem Events darstellen, gewinnt auch das Event Processing an zunehmender Bedeutung.[3] Als DBMS, welches für die Speicherung von Events in einem Event-Sourcing-System geeignet ist, sind Event Stores zu erwähnen. Anstelle des Speicherns eines aktuellen Zustands, werden bei einem Event-Sourcing-System alle Änderungen eines Zustands als Folge von Events erfasst.[4]

Vor diesem Hintergrund werden im Folgenden TSDBs und Event Stores miteinander verglichen, mit dem Ziel, die Gemeinsamkeiten und Unterschiede der beiden Arten von DBMS herauszustellen. Zumal Event Daten, vorausgesetzt, dass sie im Zeitverlauf erfasst werden, immer auch Time Series Daten darstellen, wird als Teil des Vergleichs anhand eines konkreten Anwendungsfalls untersucht, ob sich TSDBs und Event Stores gleichermaßen für die Speicherung und in einem zweiten Schritt auch für die Abfrage von Zeitreihendaten eignen.

1.2 Aufbau der Arbeit

In Kapitel 2 werden zunächst theoretische Grundlagen zu den beiden DBMS Arten, nämlich TSDBs und Event Stores, vorgestellt. Diese beginnen jeweils mit der Definition von TSDBs und Event Stores (Kapitel 2.1.1 und 2.2.1). Hier werden außerdem Besonderheiten im Vergleich zu anderen DBMS hervorgehoben. Daraufhin werden die Daten, die in den jeweiligen DBMS gespeichert werden, nämlich Time Series Daten und Event Daten, näher beleuchtet und anhand von Beispielen erklärt (Kapitel 2.1.2 und

[1] Vgl. Kholod et al. (2017), S. 97.

[2] Vgl. Bader u. a. (2017), S. 249.

[3] Vgl. Seidemann/Seeger (2017), S. 144.

[4] Vgl. Rothsberg (2015), S. iii.

2.2.2). Darüber hinaus wird der Unterschied zu anderen Arten von Daten verdeutlicht. Als jeweils letztes Kapitel der theoretischen Grundlagen zu den beiden DBMS werden ihre Anwendungsfelder mit konkreten Beispielen aufgezeigt (Kapitel 2.1.3 und 2.2.3).

In Kapitel 3 wird daraufhin ein theoretischer Vergleich der beiden DBMS Arten durchgeführt. Dieser gliedert sich zum einen in einen Vergleich anhand funktionaler und zum anderen anhand nicht-funktionaler Kriterien. Als funktionale Kriterien werden die Daten (Kapitel 3.1.1), das Speichersystem (Kapitel 3.1.2), die Performance (Kapitel 3.1.3) und zuletzt Funktionen (Kapitel 3.1.4) betrachtet, während Usability (Kapitel 3.2.1) und Support (Kapitel 3.2.2) die nicht-funktionalen Kriterien darstellen.

Die Kapitel 2 und 3 beinhalten den theoretischen Teil der Arbeit. Da die beiden Kapitel bereits als Projektarbeit als Prüfungsleistung eingereicht wurden, sind sie im Rahmen der Masterarbeit nicht erneut zu bewerten, sondern werden lediglich zur Vollständigkeit des Themas aufgeführt.

Kapitel 4 dient als Vorbereitung für Kapitel 5, in dem ein praktischer Vergleich einer ausgewählten TSDB und einem Event Store durchgeführt wird. In Kapitel 4.1 werden zunächst die Gegebenheiten beschrieben, unter welchen der praktische Vergleich der beiden DBMS stattfindet. Dies umfasst sowohl die für den praktischen Vergleich zur Verfügung stehende Hardware sowie das Betriebssystem (Kapitel 4.1.1) als auch die Daten, die im Rahmen des praktischen Vergleichs in den beiden DBMS gespeichert und abgefragt werden sollen (Kapitel 4.1.2). Aus diesen Gegebenheiten des Anwendungsfalles werden schließlich Anforderungen abgeleitet und priorisiert (Kapitel 4.1.3). Unter Berücksichtigung dieser Anforderungen werden im darauffolgenden Kapitel 4.2 aus einer Auswahl an TSDBs und Event Stores jeweils ein DBMS für den praktischen Vergleich ausgewählt, welches die zuvor definierten Anforderungen bestmöglich erfüllt. Im Kapitel 4.3 werden zum einen bereits existierende Benchmarks für TSDB und Event Stores aufgeführt. Zum anderen wird das Vorgehen beim praktischen Vergleich der ausgewählten TSDB mit dem ausgewählten Event Store beschrieben.

Inhalt des Kapitel 5 ist der praktische Vergleich der im vorherigen Kapitel 4.2 ausgewählten TSDB *InfluxDB* und des ausgewählten Event Store *Event Store*. Zum einen wird in Kapitel 5.1 unter dem Überbegriff Implementierung verglichen, wie sich die beiden DBMS installieren, konfigurieren und betreiben lassen. Darüber hinaus wird bewertet, wie die für den praktischen Vergleich zur Verfügung stehenden Daten aufbereitet werden müssen, um in die DBMS geschrieben werden zu können und wie dieser Schreibprozess umgesetzt werden kann. Zum anderen wird in Kapitel 5.2 untersucht, wie die in den beiden DBMS gespeicherten Daten abgefragt werden können. Im Fazit wird schließlich die Frage beantwortet, ob sich TSDBs und Event Stores gleichermaßen für die Speicherung und die Abfrage von Zeitreihendaten eignen.

2 Theoretische Grundlagen

2.1 Grundlagen zu Time Series Databases

Im Folgenden werden zunächst Time Series Databases definiert. Daraufhin werden Grundlagen zu Time Series (Zeitreihen) bzw. Time Series Daten (Zeitreihendaten) vorgestellt bevor Anwendungsfelder dieser Art des DBMS aufgeführt werden.

2.1.1 Time Series Database

Der Begriff Time Series Database (TSDB) wird in dieser Arbeit als Synonym für Time Series Database Management System verwendet, zumal damit nicht nur die Datenbank als Sammlung von Daten, sondern auch die Software, mit der die Daten in der Datenbank verarbeitet, verwaltet und analysiert werden, gemeint ist.[5] Diese Art des DBMS entstand als Reaktion auf die in der Einleitung (Kapitel 1.1) beschriebenen steigenden Mengen an Zeitreihendaten und aufgrund des damit einhergehenden wachsenden Bedarfs, diese Daten zu analysieren und schließlich Erkenntnisse bzw. Nutzen daraus abzuleiten. Allgemein gefasst lassen sich TSDBs als DBMS definieren, die auf die Speicherung und Abfrage von Zeitreihendaten spezialisiert sind.[6] Um diese großen Datenmengen speichern und mit einer genügend hohen Geschwindigkeit aufnehmen zu können, verfügen sie über die notwendige Skalierbarkeit.[7]

Neben dieser allgemeinen Definition lassen sich in der Literatur auch detailliertere Erläuterungen zu TSDBs finden. Laut *Bader* charakterisieren sich TSDB daran, dass sie:

(1) einen Datensatz bestehend aus Zeitstempel (timestamp), Wert und optionalen Tags speichern sowie

(2) mehrere zusammengruppierte Datensätze (z.B. gruppiert über eine Metrik oder eine Zeitreihe) speichern können,

(3) Datensätze abfragen und

(4) Abfragen in TSDB Zeitstempel oder -intervalle enthalten können.[8]

Darüber hinaus soll eine TSDB auch die Speicherung von mehr als einem Wert mit unterschiedlichen Tags pro Zeitstempel ermöglichen. Diese Definition von *Bader* verdeutlicht, dass es sich dann um eine TSDB handelt, wenn das DBMS für Zeitreihen optimiert ist. Dadurch ist es möglich, Abfragen nach Zeitstempeln oder -intervallen auszuführen, ohne die Daten dafür in eine andere Struktur modellieren zu müssen.[9]

[5] Vgl. Bader (2016), S. 12 und Härder/Rahm (2001), S. 4 f.

[6] Vgl. Bader u. a. (2017), S. 249.

[7] Vgl. Dunning/Friedman (2014), S. 10.

[8] Vgl. Bader (2016), S. 25 und Bader u. a. (2017), S. 251.

[9] Vgl. Bader (2016), S. 25.

2.1.2 Time Series (Daten)

Eine Zeitreihe (time series) ist eine Folge von Datenpunkten, die durch das wiederholte Messen von Parametern im Zeitablauf erfasst werden. Die gemessenen Werte werden zusammen mit den Zeitpunkten, zu denen die Messungen aufgenommen wurden, gespeichert. Obwohl die Messungen zumeist in regelmäßigen Zeitabständen durchgeführt werden (bspw. alle fünf Minuten), ist die Regelmäßigkeit keine zwingende Voraussetzung.[10] Wenn einer Zeitreihe für eine unbegrenzte Zeit kontinuierlich neue Datenpunkte hinzugefügt werden, wird auch von einer Streaming-Zeitreihe (streaming time series) gesprochen.[11] Diese Art von Daten entsteht insbesondere durch die in der Einleitung geschilderte Entwicklung der Industrie 4.0. bzw. von IoT-Technologien, wodurch die Anzahl von Sensoren, die regelmäßig definierte Zustände messen, stetig steigt. Darüber hinaus stellen u.a. auch Ereignisströme von Web- und Mobile-Applikationen sowie DevOps Monitoring Daten Zeitreihendaten dar.[12]

Wie zuvor beschrieben sind TSDBs darauf spezialisiert, solche Zeitreihen zu speichern. Jede Zeitreihe verfügt i.d.R. über einen Namen sowie über mehrere Zeilen von Zeitreihendaten. Eine Zeile von Zeitreihendaten beinhaltet dabei einen Zeitstempel, einen Wert und optionale Tags, die aus Namen und Werten bestehen können (beide zumeist alphanumerisch). Der Zeitstempel identifiziert einen bestimmten Zeitpunkt und wird meistens in Millisekunden beginnend am 01.01.1970 angegeben. Der Zeitraum zwischen zwei Zeitstempeln wird als Zeitspanne (time range) definiert und kann auch den Wert Null haben. Die kleinstmögliche Zeitspanne zwischen zwei Zeitstempeln, mit der die Zeitreihendaten in einer TSDB gespeichert werden, wird als Granularität bezeichnet. Hierbei ist zu beachten, dass die Granularität der Datenspeicherung von der Granularität für Abfragen abweichen kann.[13] Die Höhe der Granularität ist von der Art der Zeitreihendaten sowie den Anforderungen an die Datenanalyse abhängig, zumal nicht jede Zeitreihe auf der gleichen Detaillierungsebene gemessen werden muss, um daraus wertvolle Erkenntnisse gewinnen zu können.[14] Außerdem ist bei der Wahl der Granularität zu beachten, dass deren Höhe die Größe und Leistungsfähigkeit des DBMS stark beeinflusst. Zusätzlich ist auch eine Interpolation der Datenwerte möglich. In solchen Fällen muss neben den bereits genannten Granularitäten auch eine Interpolations-Granularität bestimmt werden, die folglich niedriger ist als die Granularität, in der die Daten aufgenommen werden.[15]

[10] Vgl. Dunning/Friedman (2014), S. 5 und S. 25.

[11] Vgl. Gao/Wang (2005), S. 1320.

[12] Vgl. Kulkarni (2017).

[13] Vgl. Bader u. a. (2017), S. 251.

[14] Vgl. Bader (2016), S. 12 f.

[15] Vgl. Shoshani (2009), S. 2994 f.

Als Beispiel von Zeitreihendaten kann die halbstündliche Messung der Temperatur in mehreren Räumen eines Hauses genannt werden. In diesem Beispiel ist die Granularität 30 Minuten. Die Werte eines Tags mit der Bezeichnung „Raum“ können dann weiter spezifizieren, aus welchem Raum des Hauses die gemessene Temperatur (Wert der Zeitreihe) kommt. Datenzeilen können schließlich über solche Tags gruppiert werden. Der Name der Zeitreihe wäre in diesem Beispiel „Temperaturmessung“. Auch mit deren Namen können Zeitreihendaten gruppiert werden und zwar auf einer gröberen Ebene als über Tags.[16]

Zeitreihendaten differenzieren sich von anderen Datensätzen dadurch, dass sie i.d.R. als neuer Eintrag in einer TSDB hinzugefügt werden und bereits gespeicherte Einträge somit nicht überschrieben werden. Ausnahmefälle entstehen nur durch die Korrektur von fehlerhaften Daten, bspw. bedingt durch verspätete Messungen oder einem Ausfall von Sensoren. Das Hinzufügen neuer Datenreihen anstelle des Überschreibens bestehender Einträge zur Abbildung des aktuellen Zustands hat den großen Vorteil, dass jegliche Veränderungen im Zeitablauf rückverfolgt werden können, zumal alle vergangenen Zustände gespeichert werden. Damit ist es möglich, vergangene Veränderungen abzubilden, aktuelle Veränderungen zu beobachten und sogar zukünftige Entwicklungen zu prognostizieren.[17]

2.1.3 Anwendungsfelder von Time Series Databases

Generell ist festzustellen, dass die Nutzung von Zeitreihendaten vor dem Hintergrund neuer Technologien, die insbesondere durch Sensoren immer mehr Zeitreihendaten generieren, zunehmend wichtiger wird und dass die Anwendungsfelder von Zeitreihendaten und somit auch von TSDBs eine große Vielfalt aufweisen.[18] Der Einsatz von TSDBs ist immer dann sinnvoll, wenn durch die Speicherung und Analyse von Zeitreihendaten Vorteile entstehen, bspw. durch die Erkennung von Mustern oder Trends.[19] Zumal Zeitreihendaten verschiedener Parameter durch den genauen Zeitpunkt der Messung (timestamp) miteinander in Verbindung gebracht werden können, ist es darüber hinaus möglich, Korrelationen zwischen bestimmten Ereignissen oder Verhaltensweisen aufzuzeigen.[20] Außerdem eignen sich Zeitreihendaten zur Anomalie-Erkennung sowie für Predictive Analytics.[21]

[16] Vgl. Bader u. a. (2017), S. 251.

[17] Vgl. Kulkarni (2017).

[18] Vgl. Dunning/Friedman (2014), S. 3.

[19] Vgl. Bader (2016), S. 12 und Dunning/Friedman (2014), S. 7.

[20] Vgl. Dunning/Friedman (2014), S. 5.

[21] Vgl. Dunning/Friedman (2014), S. 14.

Als konkrete Anwendungsfelder sind z.B. Business Intelligence Tools zur Verfolgung wichtiger Kennzahlen bzgl. der generellen Lage eines Unternehmens oder Asset-Tracking-Applikationen für Fahrzeuge oder Palletten zu nennen.[22] TSDBs werden außerdem für Finanzhandelssysteme, sowohl von klassischen Wertpapieren als auch von Kryptowährungen, eingesetzt, zumal der genaue Zeitpunkt einer Transaktion, insbesondere aufgrund der hochfrequentierten und stark fluktuierenden Daten, für Banken und Wertpapierbörsen erfolgskritisch ist.[23] Ein weiterer wichtiger Anwendungsbereich sind darüber hinaus Monitoring-Systeme. Hier ist zum einen das Monitoring von Softwaresystemen, wie bei virtuellen Maschinen, Applikationen oder Services, und zum anderen das Monitoring von physischen Systemen, bspw. Maschinen, Fahrzeuge, Häuser (Smart Home) sowie das Monitoring der gesamten Umwelt aufzuführen.[24] Werden dabei kritische bzw. entscheidende Zeitreihendaten inklusive deren Zeitpunkt der Messung erfasst, kann durch deren Speicherung und Auswertung mithilfe von TSDBs sogar die Reduktion eines Sicherheitsrisikos erreicht werden, wie es durch die Messungen der vielzähligen Sensoren bei Flugzeugen der Fall ist.[25]

2.2 Grundlagen zu Event Stores

Bevor in den folgenden Kapiteln näher auf Event Stores, Events und deren Anwendungsfelder eingegangen wird, ist es zunächst wichtig, das Konzept des Event Sourcing als dessen Basis vorzustellen, das im Zusammenhang der Softwareentwicklung zu betrachten ist. Die Kernidee des Event Sourcing besteht darin, dass nicht nur der aktuelle Zustand, sondern alle Veränderungen des Zustands einer Anwendung bzw. eines Objekts als Abfolge von Events gespeichert werden. Abfragen der Events ermöglichen dann in einem weiteren Schritt die Abbildung vergangener Zustände der Applikation bzw. des Objekts.[26] Dieses Architekturkonzept wird der dynamischen und sich ständig verändernden Umwelt gerecht, in der auch Applikationen einem kontinuierlichen Wandel unterliegen.[27] Dabei ist zu beachten, dass es sich dabei um keine Plug-and-Play-Lösung, sondern um eine Architekturentscheidung handelt, die das gesamte System, insbesondere die Datenzugriffsschicht, stark beeinflusst.[28]

Darüber hinaus kommen Events generell im Kontext von Event-Driven-Architectures vor. Dieses Architekturkonzept ermöglicht das Erkennen von und folglich das intelligen-

[22] Vgl. Kulkarni (2017).

[23] Vgl. Kulkarni (2017) und Dunning/Friedman (2014), S. 14 f.

[24] Vgl. Kulkarni (2017).

[25] Vgl. Dunning/Friedman (2014), S. 3.

[26] Vgl. Fowler (2005).

[27] Vgl. Ye (2017), S. 1.

[28] Vgl. Rybicki (2018), S. 48.

te Reagieren auf Events.[29] Da der Fokus dieser Arbeit jedoch auf Event Stores als DBMS liegt, wird nicht näher auf Event-Driven-Architectures eingegangen.

2.2.1 Event Store

Ein Event Store ist eine Datenbank zur Speicherung von Events in einem Event-Sourcing-System. Im Gegensatz zur üblichen Art der Datenhaltung wird in einem Event Store nicht der Ist-Zustand gespeichert, sondern es werden alle im Event-Sourcing-System erfassten Events, die bspw. den Zustand eines Objektes verändern, abgelegt.[30] Der Ist-Zustand eines Objektes kann in einem Event Store abgeleitet werden, indem alle Events, die den Zustand des Objekts zuvor verändert haben, in der Reihenfolge ihres Eintreffens wiedergegeben werden.[31] Damit nicht immer alle Events wiederholt werden müssen, um einen bestimmten Zustand, i.d.R. den Ist-Zustand, eines Objekts wiederherzustellen, können sogenannte Snapshots erstellt werden. Ein Snapshot stellt den Objektzustand zu einem definierten Zeitpunkt dar.[32] Sie können entweder in festen Zeitabständen oder aber nach einer festgelegten Anzahl von Events erstellt werden. Werden Snapshots verwendet, müssen folglich nur die Events, die zeitlich nach dem erzeugten Snapshot liegen, aus dem Event Store geladen und wiederholt werden, um den Ist-Zustand des Objekts wiederherzustellen. Dies führt insbesondere bei einer hohen Anzahl von Events zu einer Performanceoptimierung.[33]

Eine weitere Besonderheit eines Event Stores im Vergleich zu anderen DBMS ist außerdem, dass in einen Event Store lediglich Events hinzugefügt und keine bereits gespeicherten Event Daten gelöscht oder geändert werden. Diese Eigenschaft einer Datenbank wird auch als „append-only“ bezeichnet. Einzelne gespeicherte Datenobjekte gelten somit als unveränderbar.[34]

2.2.2 Event (Daten)

Im Kontext von Event Sourcing stellt ein Event generell eine Veränderung des Zustands eines Systems, eines Objekts oder einer Applikation dar. Dabei ist zu beachten, dass diese Zustandsveränderungen bereits in der Vergangenheit stattgefunden haben, sodass Events immer in der Vergangenheitsform beschrieben werden. Beispiele hierfür sind `PurchaseMade` oder `SeatReserved`.[35] Ein Event kann dabei auch eine Aktion

[29] Vgl. Taylor u. a. (2009), S. 1.

[30] Vgl. Rothsberg (2015), S. iii.

[31] Vgl. Müller (2016), S. 4.

[32] Vgl. Müller (2016), S. 6.

[33] Vgl. Young (2010b), S. 35.

[34] Vgl. Rybicki (2018), S. 48.

[35] Vgl. Müller (2016), S. 6.

sein, die nicht eingetreten ist, aber den Zustand eines Objekts dennoch verändert hat.[36] Der Vergangenheitsbezug von Events impliziert, dass bereits erfolgte Events selbst nicht rückgängig gemacht und auch nicht verändert werden können.[37] Allerdings können die durch bereits geschehene Events eingetroffenen Zustände durch nachfolgende Events geändert oder negiert werden. Hier ist bspw. das Event `SeatReservationCancelled` zu nennen, welches das Ergebnis des zuvor genannten Reservierungs-Events (`SeatReserved`) wieder aufhebt.[38] Daran wird ebenfalls deutlich, dass alle Events, mit Ausnahme vom Ersten, stets auf dem jeweils zeitlich vorherigen Event aufbauen.[39] Eine solche Abfolge von Events, die nach ihrem Eintreten geordnet ist, wird als Event Stream bezeichnet und kann unendlich lang sein.[40]

Eine weitere Eigenschaft von Events ist, dass sie als atomar und verzögerungsfrei gelten, d.h. sie treten ein oder nicht.[41] Darüber hinaus sind sie als one-way-Nachrichten zu sehen, da sie von einer einzigen Quelle generiert, jedoch von mehreren Empfängern erhalten werden können.[42] Softwaremodule und Sensoren sind Beispiele für eine solche Quelle von Events, die auch Event Source, Event Producer oder Event Publisher genannt wird. Als Event Empfänger sind ebenfalls Softwaremodule aber auch Datenbanken, wie Event Stores, zu nennen. Sie werden außerdem als Event Consumer, Event Listener oder Event Sink bezeichnet.[43] Eine Voraussetzung für Event Empfänger ist, dass sie das Eintreten der für sie relevanten Events erkennen müssen, um sie im Beispiel eines Event Stores als Datenbank speichern zu können.[44]

Ein Event, als eine in der Vergangenheit geschehene Zustandsveränderung eines Systems, eines Objekts oder einer Applikation, ist nicht mit den eigentlichen Daten, mit denen ein Event beschrieben wird, gleichzusetzen. Vor diesem Hintergrund, sind es streng genommen nicht Events, wie zuvor erläutert, sondern Event Daten bzw. Event Objects, die in Event Stores gespeichert werden.[45] Zu diesen Event Daten zählt i.d.R. zum einen eine eindeutige Event ID und zum anderen weitere Metadaten des Events, wie der Event Typ, die Event Source und der Zeitpunkt des Eintretens des Events in Form eines Zeitstempels.[46] Da die Begriffe Events und Event Daten bzw. Event Ob-

[36] Vgl. Taylor u. a. (2009), S. 14.

[37] Vgl. Event Store LLP (2018c) und Betts u. a. (2012), S. 235.

[38] Vgl. Betts u. a. (2012), S. 235.

[39] Vgl. Müller (2016), S. 6.

[40] Vgl. Hong et al. (2009), S. 1030.

[41] Vgl. Ericson et al. (2009), S. 1044.

[42] Vgl. Betts u. a. (2012), S. 235.

[43] Vgl. Luckham/Schulte (2011) und Taylor u. a. (2009), S. 16 f.

[44] Vgl. Taylor u. a. (2009), S. 17.

[45] Vgl. Chandy (2009), S. 1041.

[46] Vgl. Dunkel u. a. (2008), S. 125.

jects in der Literatur jedoch häufig synonym verwendet werden, umfasst ein Event auch in dieser Arbeit beide Definitionen.

Neben der Event Bezeichnung bzw. Definition, der Event ID sowie den Metadaten, kann ein Event auch zusätzliche Informationen über die Zustandsveränderung, d.h. Event-spezifische Daten, enthalten, welche für die Verarbeitung und weiterführenden Analyse der Event Daten notwendig sind.[47] Bspw. kann das Event mit dem Namen `SeatReserved` den detaillierteren Zusatz „Sitz 11C wurde von Lisa gebucht“ enthalten. Diese Zusatzinformationen sollten, wie das aufgeführte Beispiel verdeutlicht, die Intention des Events im Business-Kontext mit dafür typischen Ausdrücken schildern und nicht zu technisch sein, wie hingegen „in der Buchungstabelle wurde das Feld Name der Zeile mit dem Key 11C auf den Wert Lisa geändert“.[48]

2.2.3 Anwendungsfelder von Event Stores

Als konkretes Anwendungsfeld eignen sich Event Stores bspw. für Audit-Protokolle bzw. Revisionsaufzeichnungen, v.a. im Finanzbereich. Dies ist durch die „append-only“-Eigenschaft des Event Stores und dem unveränderbaren Charakter von Events zu begründen, zumal Unternehmen zur Aufbewahrung bestimmter Daten verpflichtet sind. Darüber hinaus begünstigt die Eigenschaft der Rückverfolgbarkeit, die durch die Speicherung jedes Events und somit jeder Veränderung des Zustands eines Objektes bzw. einer Applikation entsteht, die Lokalisierung von Fehlern bzw. das Debuggen. Die Entstehung von Fehlern bzw. Bugs kann dank der Tatsache, dass jeder Zustand durch die Wiederholung von Events wiederhergestellt werden kann, leichter rekonstruiert werden.[49]

Beide Beispiele verdeutlichen den informativen Wert, der durch die Speicherung jeglicher Veränderungen in Form von Events entsteht und von großer Bedeutung für ein Unternehmen sein kann. Insbesondere bei Systemen, welche die menschliche Interkation involvieren, wie ein Online Shop, ist zu Beginn nicht immer bekannt, welche Systemzustände oder Interaktionen zu einem späteren Zeitpunkt analysiert werden müssen oder sollen. Wird hier ein Event-Sourcing-System verwendet, kann eine erst zu einem späteren Zeitpunkt aufkommende Frage, z.B. welche Artikel vor dem Bezahlvorgang aus dem Warenkorb entfernt werden, mithilfe einer Abfrage der Event Daten im Event Store beantwortet werden, da durch die Wiederholung aller im Event Store gespeicherten Events (z.B. `ArticleAdded` und `ArticleRemoved`) jeder Zustand eines Warenkorbs rekonstruiert werden kann.[50]

[47] Vgl. Betts u. a. (2012), S. 235 und Dunkel u. a. (2008), S. 125.

[48] Vgl. Betts u. a. (2012), S. 235.

[49] Vgl. Müller (2016), S. 5.

[50] Vgl. Müller (2016), S. 5.

Darüber hinaus werden Event-Sourcing-Systeme inklusive Event Stores häufig im Zusammenhang mit der sogenannten Command Query Responsibility Segregation (CQRS) eingesetzt. Dabei ist hervorzuheben, dass Event Sourcing keine zwingende Voraussetzung für CQRS ist. Die Idee hinter CQRS ist, dass die write/command- und die read/query-Seite eines Systems voneinander getrennt sind.[51] Bei der Kombination von Event Sourcing und CQRS wird ein Event Store i.d.R. auf der write/command-Seite genutzt, um die dort erzeugten Events (Veränderungen eines Zustands) zu speichern. Zumal alle Abfragen auf der read/query-Seite gemacht werden, wird die Performance des Event Stores nicht negativ beeinflusst. Die auf der write/command-Seite erzeugten Events werden mittels asynchroner Nachrichten zur read-Seite gesendet, die nach CQRS eine eigene Datenbank hat. In dieser zweiten Datenbank wird der Zustand des Objekts abgebildet, sodass dieser abgefragt werden kann. Dabei kann ggf. nur eine sogenannte eventual consistency gewährleistet werden, wenn die Datenbank auf der read/query-Seite (noch) nicht genau die gleichen Daten enthält, wie der Event Store auf der write/command-Seite.[52] Laut *Young* bietet Event Sourcing jedoch sogar den Vorteil, dass ein Event Store als einzige Datenbank, sowohl auf der write/command-Seite zum Speichern von Events als auch auf der read/query-Seite für Abfragen, eingesetzt werden kann. Positiv in diesem Fall ist, dass keine Synchronisation zwischen verschiedenen Datenbänken und -modellen stattfinden muss.[53]

[51] Vgl. Ye (2017), S. 8.

[52] Vgl. Overeem et al. (2017), S. 2.

[53] Vgl. Young (2010a).

3 Theoretischer Vergleich von Time Series Databases und Event Stores

3.1 Funktionale Kriterien

Im Folgenden werden TSDBs und Event Stores nach ausgewählten funktionalen Kriterien für DBMS miteinander verglichen. Zunächst werden die Gemeinsamkeiten und Unterschiede von Time Series Daten und Event Daten, die in den beiden DBMS gespeichert werden, aufgeführt. Danach werden die verschiedenen Speichersysteme von TSDBs und Event Stores vorgestellt und jeweils konkrete Beispiele genannt. Anschließend wird beschrieben, wie TSDBs und Event Stores aufgrund ihrer Performance und ihrer angebotenen Funktionen Vorteile gegenüber traditionellen DBMS bieten.

3.1.1 Daten

Eine Gemeinsamkeit und die hervorzuhebende Besonderheit von Time Series Daten und Event Daten sind, dass die Speicherung dieser Daten die Analyse und das Monitoring jeglicher Veränderungen im Zeitablauf erlaubt und somit einen großen informativen Mehrwert gegenüber anderen Arten von Daten bringt, mit denen lediglich ein aktueller Zustand abgebildet werden kann.[54] Einige Autoren, wie *Kulkarni*, geben sogar an, dass Event Daten gleichzeitig auch Time Series Daten sind. *Kulkarni* verdeutlicht dies mit dem Beispiel einer Webanwendung, bei der jeder Login eines Users als separates Event im Zeitablauf gespeichert wird. Durch die Speicherung dieser Daten kann das vergangene Login-Verhalten des Users nachgebildet und folglich auch analysiert werden.[55] Zumal die Events in diesem Beispiel im Zeitablauf gespeichert werden und Zeitreihendaten in Kapitel 2.1.2 als Daten von im Zeitablauf wiederholten Messungen definiert werden, ist es richtig zu sagen, dass Events auch Zeitreihen darstellen.

Als weitere Gemeinsamkeit gilt der „append-only“-Charakter beider Arten von Daten, d.h. sie werden lediglich zur entsprechenden Datenbank hinzugefügt und i.d.R. weder verändert noch gelöscht.[56] Darüber hinaus werden beide Arten von Daten zusammen mit einem Zeitstempel (timestamp) im jeweiligen DBMS gespeichert.[57] Während die Zeitstempel bei Zeitreihendaten üblicherweise in Millisekunden beginnend am 01.01.1970 angegeben werden und sie den genauen Zeitpunkt angeben, zu dem die Zeitreihendaten gemessen wurden (vgl. Kapitel 2.1.2), sind bei Events verschiedene timestamps zu unterscheiden. Zum einen kann der Zeitstempel eines Events den Zeitpunkt identifizieren, zu dem das Event in einem System (Event Quelle) erzeugt wurde.

[54] Vgl. Kulkarni (2017) und Rybicki (2018), S. 48.

[55] Vgl. Kulkarni (2017).

[56] Vgl. Kulkarni (2017) und Rybicki (2018), S. 48.

[57] Vgl. Bader u. a. (2017), S. 251 und Hong et al. (2009), S. 1030.

Zum anderen kann durch den Zeitstempel auch der Zeitpunkt abgebildet werden, zu dem ein Event in einem anderen System (Event Empfänger) eingegangen ist. Die Zeitstempel eines Events können sich daher auf unterschiedliche Uhren von verschiedenen Systemen beziehen.[58]

Neben den aufgeführten Gemeinsamkeiten von Time Series Daten und Event Daten können aber auch Unterschiede herausgestellt werden. Zwar ist es keine zwingende Voraussetzung von Zeitreihendaten, vgl. auch Kapitel 2.1.2, jedoch ist es üblich, dass Zeitreihendaten in regelmäßigen Zeitabständen gemessen und somit auch in regelmäßigen Zeitabständen in einem DBMS (TSDB) gespeichert werden.[59] Im Gegensatz dazu finden Events i.d.R. nach unvorhersehbaren Mustern statt, wie auch das Beispiel des User-Logins in einer Webanwendung verdeutlicht. Demzufolge erfolgt die Speicherung von Event Daten eher in unregelmäßigen Zeitabständen.[60]

3.1.2 Speichersystem

Wie in Kapitel 2.1 beschrieben, sind TSDBs darauf spezialisiert, Zeitreihendaten zu speichern. Je TSDB kann die Speicherung der Daten in unterschiedlichen Systemen erfolgen. Einige Fragen, die sich bei der Wahl des Speichersystems zu stellen sind, lauten z.B. wie viele verschiedene Zeitreihen gespeichert werden sollen, welche Art von Zeitreihendaten zu speichern ist, mit welcher Granularität die Daten gesammelt werden und zu speichern sind und für wie lange die Zeitreihendaten gespeichert werden sollen.[61]

Vor dem Hintergrund, dass TSDBs über verschiedene Speichersysteme verfügen können, hat *Bader* bestehende TSDBs in drei Gruppen eingeteilt:

(1) Erfordernis eines NoSQL[62] DBMS

(2) Keine Abhängigkeit von anderen DBMS

(3) Relationale Datenbank Management Systeme (RDBMS) als TSDB.[63]

In der ersten Gruppe befinden sich TSDBs, die ein bestehendes NoSQL DBMS, wie bspw. *Cassandra*, *HBase* oder *CouchDB*, für die Speicherung der Time Series Daten benötigen.[64] TSDBs, die zu dieser Gruppe gehören, sind u.a. *OpenTSDB* und

[58] Vgl. Luckham/Schulte (2011).

[59] Vgl. Seidemann/Seeger (2017), S. 145 und Shoshani (2009), S. 2995.

[60] Vgl. Shoshani (2009), S. 2995.

[61] Vgl. Dunning/Friedman (2014), S. 25 f.

[62] NoSQL steht im Deutschen für „Not only SQL“, wobei SQL die Abkürzung für Structured Query Language ist. NoSQL DBMS sind Datenbanksysteme, die einen nicht-relationalen Ansatz verfolgen (vgl. Luber/Litzel (2017b)).

[63] Vgl. Bader (2016), S. 26 und Bader u. a. (2017), S. 152.

[64] Vgl. Bader (2016), S. 27.

KairosDB. *OpenTSDB* verwendet das Open-Source-DBMS *HBase*, um Zeitreihendaten zu speichern und abzufragen.[65] *OpenTSDB* nennt sich „The Scalable Time Series Database“[66] und wirbt damit, große Mengen an Zeitreihendaten speichern zu können, ohne dabei Granularität zu verlieren.[67] Währenddessen bezeichnet sich *KairosDB* als „Fast Time Series Database on *Cassandra*“[68] und nutzt demzufolge *Cassandra* als Speichersystem.

Die zweite Gruppe umfasst all diejenigen TSDBs, die Zeitreihendaten unabhängig von einem weiteren DBMS speichern können und somit ein speziell für die TSDB entwickeltes Speichersystem besitzen. Zumal Metadaten und andere zusätzliche Informationen nicht direkt als Zeitreihendaten gelten, können diese auch bei dieser Gruppe von TSDBs in einem anderen DBMS gespeichert werden.[69] Als Beispiel einer TSDB dieser zweiten Gruppe ist *InfluxDB* zu nennen. *InfluxDB* ist ein leistungsstarkes DBMS, welches speziell für Zeitreihendaten bzw. Daten mit Zeitstempel entwickelt wurde. Abfragen der Daten erfolgen hierbei mit der Influx Query Language (InfluxQL), einer SQL-ähnlichen Abfragesprache.[70] Ein weiteres Beispiel für eine TSDB ohne Abhängigkeit zu einem anderen DBMS ist *Druid*. *Druid* ist ebenfalls für Daten mit Zeitstempel optimiert. Zumal *Druid* Daten anhand des Faktors Zeit partitioniert, können Abfragen, die einen Zeitfilter enthalten, deutlich schneller ausgeführt werden als Abfragen ohne Zeitfilter.[71]

Zur dritten Gruppe von TSDBs gehören RDBMS, mit denen Zeitreihendaten gespeichert werden können. Auch wenn diese RDBMS Abhängigkeiten zu anderen DBMS haben, werden sie dieser dritten Gruppe zugeordnet.[72] *PostgreSQL* ist ein solches RDBMS, welches die Speicherung von Zeitreihendaten ermöglicht. Bei *PostgreSQL* bestehen keine Abhängigkeiten zu anderen DBMS.[73] In diesem Zusammenhang ist ebenfalls *TimescaleDB* als eine Erweiterung von *PostgreSQL* zu nennen. *TimescaleDB* wurde speziell für Zeitreihendaten optimiert, da auch bei diesem DBMS Daten automatisch zeitbezogen partitioniert werden. *TimescaleDB* unterstützt, wie auch *PostgreSQL*, SQL als Abfragesprache und stellt darüber hinaus zusätzliche Funktionen zur Analyse und Manipulation von Zeitreihendaten zur Verfügung.[74]

Genau wie Time Series Daten in TSDBs können auch Event Daten in Event Stores in unterschiedlichen Systemen gespeichert werden. Das Speichersystem ist dabei frei

[65] Vgl. The OpenTSDB Authors (2010a-2019).

[66] The OpenTSDB Authors (2010b-2019).

[67] Vgl. The OpenTSDB Authors (2010b-2019).

[68] KairosDB (2015).

[69] Vgl. Bader u. a. (2017), S. 152.

[70] Vgl. InfluxData (2019d).

[71] Vgl. Druid (2019b).

[72] Vgl. Bader u. a. (2017), S. 152.

[73] Vgl. Bader (2016), S. 35.

[74] Vgl. Timescale (2019a).

wählbar und kann wie bei TSDBs z.B. ebenfalls ein RDBMS, ein NoSQL DBMS eine einfache Datei oder ein eigens für Event Daten konzipiertes Speichersystem sein.[75] Bei der Wahl des Speichersystems sind Kriterien wie Verfügbarkeit, Zuverlässigkeit, Konsistenz, Skalierbarkeit und Performance im konkreten Anwendungsfall zu berücksichtigen.[76]

Da Events keine komplexen Datenstrukturen aufweisen (vgl. hierzu auch Kapitel 2.2.2 und Kapitel 3.1.1), ist es i.d.R. nicht notwendig, ein RDBMS als Speichersystem für Events zu verwenden.[77] Dies wird dadurch bestärkt, dass in einem Event Store nur zwei Operationen zwingend durchgeführt werden müssen, nämlich die „append-only"-Operation, um Events im Event Store hinzuzufügen bzw. zu speichern, sowie die Abfrage aller Events, um den aktuellen Zustand eines Objekts zu bestimmen.[78] Welche Art eines NoSQL DBMS wiederum für die Speicherung von Events geeignet ist, hängt auch von den individuellen Anforderungen an einen Event Store ab. Eine solche Evaluation verschiedener Arten von NoSQL DBMS (Key-Value Store, Document Store, Column Store, Graph Database) als Speichersystem eines Event Stores ist in der Arbeit von *Rothsberg* zu finden. Aufgrund des Erfüllens aller dort definierten Anforderungen wird in *Rothsbergs* Arbeit die Graphdatenbank Neo4j für die Implementierung eines Event Stores ausgewählt.[79]

Als Beispiel eines Event Stores, der kein bestehendes DBMS zur Speicherung von unveränderbaren Events im Zeitablauf verwendet, ist *Event Store* zu nennen.[80] *Event Store* bezeichnet sich selbst als „The open-source, functional database with Complex Event Processing in JavaScript"[81]. In diesem Zusammenhang ist auf die funktionale Programmierung zu verweisen.[82] Bei der funktionalen Programmierung stellt die Liste die wichtigste Datenstruktur dar. Als entscheidende Eigenschaften sind außerdem die Trennung der Erstellung und Nutzung von Listen sowie die unveränderten Originaldaten zu nennen.[83]

Folglich kann festgehalten werden, dass sowohl die Speicherung von Time Series Daten als auch die Speicherung von Event Daten je nach TSDB bzw. Event Store in unterschiedlichen Systemen erfolgen kann, was somit eine Gemeinsamkeit der beiden DBMS Arten darstellt. Die Speichersysteme können dabei, wie beschrieben, bereits

[75] Vgl. Rothsberg (2015), S. 11.

[76] Vgl. Betts u. a. (2012), S. 98.

[77] Vgl. Betts u. a. (2012), S. 245.

[78] Vgl. Rothsberg (2015), S. 12.

[79] Vgl. Rothsberg (2015), S. 25 ff.

[80] Vgl. Event Store LLP (2018f).

[81] Event Store LLP (2018f).

[82] Vgl. Event Store LLP (2018d).

[83] Vgl. Grimm (2009).

bestehende RDBMS, NoSQL DBMS oder aber unabhängige Systeme der TSDB bzw. des Event Stores sein.

3.1.3 Performance

Speziell für Time Series und Event Daten optimierte DBMS, nämlich TSDBs und Event Stores, werden insbesondere aus dem Grund eingesetzt, da sie im Vergleich zu traditionellen DBMS eine bessere Performance bei der Speicherung und Abfrage der entsprechenden Datenarten aufweisen.[84] Nur durch diese jeweilige Optimierung ist es mit den beiden Arten von DBMS möglich, die Vielzahl an write-Operationen durchzuführen, die für die Speicherung der in der Einleitung beschriebenen steigenden Mengen von Time Series und Event Daten erforderlich sind.[85] Unter Berücksichtigung des vorherigen Kapitels 3.1.2 wird dieser gemeinsame Vorteil von TSDBs und Event Stores gegenüber traditionellen DBMS vorwiegend dann erzielt, wenn für die Speicherung der Zeitreihendaten bzw. Events ein NoSQL DBMS oder ein eigens für die TSDB bzw. den Event Store konzipiertes System eingesetzt wird, da diese die dafür notwendige Skalierbarkeit besitzen.[86] Faktoren wie das Datenformat, der Daten-Workflow und die Gestaltung der Tabellen haben dabei ebenfalls einen Einfluss auf die Performance des DBMS.[87]

Die im vorherigen Kapitel 3.1.2 vorgestellte TSDB *OpenTSDB*, die *HBase* zur Speicherung von Time Series Daten verwendet, erlangt ihre Performance-Vorteile gegenüber traditionellen DBMS bspw. dadurch, dass sie die Zeitreihendaten in hybriden Tabellen speichert. Dies bedeutet in diesem konkreten Fall, dass zum einen einzelne Datenpunkte der Time Series, sobald sie erzeugt und empfangen werden, im NoSQL Wide-Column-Store *HBase* gespeichert werden. Zum anderen werden die gespeicherten Daten im Hintergrund kontinuierlich in das sogenannte „BLOB“-Format[88] komprimiert.[89] Dadurch können sowohl komprimierte als auch nicht komprimierte Zeitreihendaten in einer Reihe der hybriden Tabelle stehen.[90] Die Umwandlung der Daten in das BLOB-Format ermöglicht nicht nur eine bessere Speicher-Effizienz, sondern schließlich auch eine höhere Geschwindigkeit bei der Datenabfrage.[91] Konkrete Vergleiche der Performance verschiedener TSDBs bei Schreib- und Lese-Operationen sind in der Arbeit von

[84] Vgl. Dunning/Friedman (2014), S. 12, Kulkarni (2017) und Betts u. a. (2012), S. 240.

[85] Vgl. Seidemann/Seeger (2017), S. 144 und Kulkarni (2017).

[86] Vgl. Dunning/Friedman (2014), S. 12 f und S. 36.

[87] Vgl. Dunning/Friedman (2014), S. 69.

[88] BLOB steht für Binary Large Object. Dabei handelt es sich um ein großes binäres Datenobjekt, das in DBMS in besonderer Form gespeichert wird (vgl. Litzel (2018)).

[89] Vgl. Dunning/Friedman (2014), S. 39 f.

[90] Vgl. Dunning/Friedman (2014), S. 31 f.

[91] Vgl. OpenTSDB (2018c) und Dunning/Friedman (2014), S. 32.

Bader oder in mehreren von *InfluxData* veröffentlichten technischen Papern zu finden.[92]

Der im vorherigen Kapitel 3.1.2 vorgestellte Event Store *Event Store* erreicht die beschriebenen Performance-Vorteile wiederum durch die „append-only"-Eigenschaft von Events. Dies ist dadurch begründet, dass Speichersysteme, bei denen lediglich Daten hinzugefügt und keine bereits gespeicherten Daten manipuliert oder gelöscht werden, üblicherweise einfacher skalieren als diejenigen DBMS, bei denen Manipulationen und Löschungen von Daten möglich sind und somit vermehrt Zugriffssperren verursachen.[93] Vor diesem Hintergrund können mit *Event Store* laut dessen Aussage etwa 15.000 Schreib- und 50.000 Lese-Operationen pro Sekunde ausgeführt werden.[94]

3.1.4 Funktionen

Da die Funktionen und Operationen, die von TSDBs und Event Stores angeboten werden, auf Time Series und Event Daten ausgerichtet sind, stellen diese zusätzlich zu der im vorherigen Kapitel beschriebenen Performance einen weiteren Grund und eine Gemeinsamkeit dar, warum TSDBs und Event Stores für die Speicherung und Abfrage von Zeitreihendaten bzw. Events traditionellen DBMS vorgezogen werden. Diese Funktionen sind bei Event Stores bspw. die (rollierende) Erstellung von Snapshots, um bei der Abfrage des aktuellen Zustands eines Objekts nicht immer alle, sondern nur die zeitlich nach dem Snapshot liegenden Events wiederholen zu müssen (vgl. Auch Kapitel 2.2.1).[95]

Bei TSDBs gehören wiederum z.B. Regeln zur (Langzeit-)Datenspeicherung, flexible Aggregationen nach dem Faktor Zeit und kontinuierlich laufende Abfragen bzw. Berechnungen von Funktionen sowie die Speicherung der dabei berechneten Daten (bspw. Durchschnitt je Stunde) zu den Funktionen bzw. Operationen, die im Kontext von Time Series Daten üblich sind.[96] Allerdings ist zu beachten, dass nicht alle TSDBs über die gleichen Funktionen und Operationen verfügen. Ein wesentlicher Unterschied zwischen einzelnen TSDBs liegt z.B. darin, dass nicht alle TSDBs für jede Art von Zeitreihendaten optimiert sind. Obwohl Time Series Daten üblicherweise in regelmäßigen Zeitabständen erfasst und gespeichert werden (vgl. Kapitel 3.1.1), gibt es auch Anwendungsfälle mit unregelmäßigen Zeitreihendaten, bspw. wenn Event Daten Zeitreihen darstellen (vgl. das Beispiel des User-Logins in einer Webanwendung aus Kapitel

[92] Vgl. Bader (2016), S. 75 ff. und InfluxData (2019b).

[93] Vgl. Event Store LLP (2018e).

[94] Vgl. Event Store LLP (2018f).

[95] Vgl. Chinchilla (2018b).

[96] Vgl. Kulkarni (2017) und Bader (2016), S. 38.

3.1.1). Im Gegensatz zu anderen TSDBs wie *OpenTSDB* ist *InfluxDB* sowohl für regelmäßige als auch für unregelmäßige Zeitreihendaten optimiert.[97]

Weitere funktionale Unterschiede zwischen den in Kapitel 3.1.2 vorgestellten TSDBs betreffen die o.g. Möglichkeit einer kontinuierlichen Berechnung von Funktionen. Damit sind Abfragen gemeint, die kontinuierlich und in Echtzeit auf den in einer TSDB eingehenden Daten ausgeführt werden. Die Ergebnisse dieser regelmäßigen Abfragen werden dabei ebenfalls in der TSDB als spezifizierte Metrik gespeichert. Als gängige kontinuierliche Abfrage ist die Berechnung eines Durchschnittswertes für eine definierte Zeitspanne zu nennen.[98] Im Beispiel „Temperaturmessung“ aus Kapitel 2.1.2 kann in der zur Speicherung der Zeitreihendaten verwendeten TSDB auf Basis der halbstündlich erfassten Temperaturmessungen bspw. automatisch eine Durchschnittstemperatur je Stunde berechnet und folglich auch in der TSDB als Wert einer Durchschnittsmetrik gespeichert werden. Die Granularität der aggregierten Durchschnittswerte ist dabei gröber als die Granularität der in der TSDB eingehenden Werte (60 Minuten gegenüber 30 Minuten). Solche kontinuierlichen Berechnungen sind nur mit den TSDBs *Druid*, *InfluxDB*, *PostgreSQL* und als Erweiterung davon auch *TimescaleDB* möglich, während *OpenTSDB* und *KairosDB* diese Funktion nicht anbieten. Ein weiterführender Vergleich dieser und weiterer TSDBs anhand Funktionen bzw. Operationen sowie zusätzlicher Kriterien ist in der Arbeit von *Bader* zu finden.[99] Folglich ist im konkreten Anwendungsfall unter den jeweils geltenden Anforderungen zu entscheiden, welche TSDB am geeignetsten ist.

3.2 Nicht-Funktionale Kriterien

In diesem Kapitel werden TSDBs und Event Stores anhand nicht-funktionaler Kriterien miteinander verglichen. Diese Kriterien betreffen nicht nur einzelne Funktionen bzw. Eigenschaften der TSDB bzw. des Event Stores, sondern das jeweilige DBMS als Ganzes. Zum einen wird die Usability von TSDBs und Event Stores gegenübergestellt. Als Faktoren, welche die Usability beeinflussen, werden die jeweils verfügbaren Benutzeroberflächen (User Interfaces (UIs)) und Erweiterungsmöglichkeiten betrachtet. Zum anderen wird der gebotene Support gängiger TSDBs und Event Stores miteinander verglichen.

[97] Vgl. Dix (2016), S. 3.

[98] Vgl. Bader (2016); S. 38 und Bang/Anderson (2018).

[99] Vgl. Bader (2016), S. 36 ff.

3.2.1 Usability

Die Usability bzw. Gebrauchstauglichkeit beschreibt das „Ausmaß, in dem ein System […] durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um festgelegte Ziele effektiv, effizient und zufriedenstellend zu erreichen“[100]. Dabei können eine geeignete Benutzeroberfläche und Erweiterungsmöglichkeiten die Usability positiv beeinflussen. Da der Nutzungskontext bei TSDBs und Event Stores i.d.R. die Speicherung, Abfrage und Analyse von Time Series bzw. Event Daten umfasst, sollten die Benutzeroberflächen und Erweiterungen der beiden DBMS Arten die Durchführung dieser drei Haupttätigkeiten bestmöglich unterstützen. Vor diesem Hintergrund stellen die einzelnen TSDBs und Event Stores UIs zur Verfügung, die für Zeitreihendaten bzw. Events optimiert sind.

Zumal Zeitreihendaten häufig in Form eines Liniendiagramms visualisiert werden, bietet die TSDB *OpenTSDB* ihren Anwendern und Anwenderinnen ein bereits eingebautes Graphical User Interface (GUI), welches die Auswahl einzelner oder mehrerer Metriken ermöglicht, die dann in Form eines Diagramms dargestellt werden. Alternativ ist ebenfalls ein HTTP Application Programming Interface (API) für die Anbindung an externe Systeme, wie Dashboards, Monitoring-Systeme oder Statistik Softwares, verfügbar.[101] Als Beispiele sind hier die Open-Source-Software für Zeitreihenanalyse *Grafana* und die Statistik Software *R* zu nennen. Eine Auflistung weiterer Systeme zur Anbindung an *OpenTSDB* ist in dessen Dokumentation zu finden.[102]

Auch die TSDB *KairosDB* verfügt über ein eigenes Web UI, mit dem Time Series Daten abgefragt und in Diagrammen angezeigt werden können. Dieses Web UI ist laut *KairosDB* jedoch hauptsächlich für die Entwicklung bzw. die Entwickler entworfen worden.[103] Zusätzlich ist aber auch bei *KairosDB* die Anbindung an Grafana mit einem schöneren und für Zeitreihendaten sehr gut geeigneten GUI möglich.[104] Bei der TSDB *Druid* hat der Anwender oder die Anwenderin neben der Nutzung des eigenen UI für eine bessere Usability ebenfalls die Möglichkeit, verschiedene externe Systeme, wie z.B. *Grafana* und *R*, anzubinden.[105] Gleichermaßen können auch bei *PostgreSQL* und *TimescaleDB* externe Softwares (z.B. *Grafana* und *Tableau*) zur Visualisierung und Analyse der Time Series Daten verwendet werden.[106]

Das wohl umfangreichste eigene Interface der in Kapitel 3.1.2 aufgeführten TSDBs mit Möglichkeiten zur Datenabfrage, zur Einstellung von Warnmeldungen (Alerts) sowie

[100] Vgl. DIN Deutsches Institut für Normung e. V. (2011).

[101] Vgl. The OpenTSDB Authors (2010a-2019).

[102] Vgl. OpenTSDB (2018a).

[103] Vgl. KairosDB (2015).

[104] Vgl. Grafana Labs (2018).

[105] Vgl. Druid (2019a).

[106] Vgl. Timescale (2019b).

zur Visualisierung der Daten bietet *InfluxDB* bzw. die *InfluxData* Plattform. Mit dem Interface *Chronograf* kann der Anwender oder die Anwenderin neben der Erstellung von individuellen Dashboards zur Visualisierung der Time Series Daten ebenfalls aus einer Auswahl von vorgefertigten Dashboards wählen, was ihm oder ihr einen schnellen und einfachen Start ermöglicht.[107] Hat ein Anwender oder eine Anwenderin jedoch schon Erfahrungen mit anderen Visualisierungstools gesammelt, ist auch bei *InfluxDB* eine Anbindung an externe Systeme, wie Grafana, möglich, was in einem solchen konkreten Fall die Usability für den jeweiligen User erhöhen kann.[108]

Der in Kapitel 3.1.2 vorgestellte Event Store *Event Store* bietet drei verschiedene Oberflächen zur Interaktion. Zum einen stellt *Event Store* ein eigenes Admin UI zur Verfügung. Zum anderen kann die Interaktion mittels einer HTTP Application Programming Interface (API) oder als dritte Möglichkeit mit einer Client API, bspw. .NET Core Client API und JVM Client, erfolgen.[109] Positiv an dieser Auswahl an Interaktionsmöglichkeiten ist, dass Anwender und Anwenderinnen individuell ein UI wählen können, mit dem sie die für sie beste Usability erreichen. Mit dem Admin UI des *Event Stores* können intuitiv Event Streams erstellt und Events hinzugefügt werden. Darüber hinaus können gesamte Event Streams oder auch einzelne Events abgefragt bzw. gelesen werden. Außerdem ermöglicht das Admin UI das Abonnieren von Event Streams, sodass der Anwender oder die Anwenderin bei Veränderungen bzw. beim Eintreffen neuer Events benachrichtigt wird (Alert-Funktion).[110] Ein Überblick über die gesamten Inhalte des *Event Store* Admin UI ist in dessen Dokumentation zu finden.[111]

Somit kann festgehalten werden, dass sowohl TSDBs als auch Event Stores für Zeitreihendaten bzw. Events optimierte Oberflächen sowie Erweiterungsmöglichkeiten anbieten und dies somit eine Gemeinsamkeit darstellt. Im konkreten Vergleich liegt der Fokus bei TSDBs durch die Anbindungsmöglichkeiten an *Grafana* o.ä. noch vielmehr in der Visualisierung der Daten. Welche TSDB oder Event Store jedoch die höchste Usability für den Anwender oder die Anwenderin verspricht, ist im individuellen Nutzungskontext zu beurteilen.

3.2.2 Support

Der Support unterteilt sich zum einen in den kommerziellen Support vom Entwickler bzw. Hersteller und zum anderen in den Support von anderen Anwendern und Anwenderinnen des jeweiligen DBMS in verschiedenen Communities, wie bspw. GitHub. Da

[107] Vgl. InfluxData (2019a).

[108] Vgl. Grafana Labs (2018).

[109] Vgl. Chinchilla (2019a).

[110] Vgl. Chinchilla (2019b).

[111] Vgl. Event Store LLP (2018a).

der Support von anderen Anwendern und Anwenderinnen nicht als eigene Leistung der jeweiligen TSDB oder des jeweiligen Event Stores betrachtet werden kann und der Community Support für alle bisher vorgestellten TSDBs und Event Stores gleichermaßen vorhanden ist, wird in diesem Kapitel hauptsächlich der kommerzielle Support der DBMS gegenübergestellt.

InfluxDB ist die einzige der sechs im Kapitel 3.1.2 vorgestellten TSDBs, die, zusätzlich zur Veröffentlichung einer ausführlichen Dokumentation auf der Webseite und dem Community Support, auch verschiedene kommerzielle Support Lösungen (für die gesamte *InfluxData* Plattform) anbietet. Dabei kann zwischen drei Support Paketen gewählt werden, die sich durch die Anzahl der inkludierten User, die angebotenen Service-Zeiten und folglich auch die Kosten unterscheiden.[112] Im Gegensatz dazu stellen die TSDBs *Druid*, *OpenTSDB* und *KairosDB* lediglich Dokumentationen auf ihren Webseiten zur Verfügung und verweisen auf diverse Communities.[113] *PostgreSQL* führt auf ihrer Webseite neben einer Dokumentation und Links zu verschiedenen Communities zwar auch die Rubrik „Professional Services“ auf, allerdings werden hier lediglich Unternehmen aufgeführt, die Beratung und Support für *PostgreSQL* anbieten, was nicht als Eigenleistung von *PostgreSQL* bewertet werden kann.[114] *TimescaleDB* veröffentlicht über die Dokumentation und den Community Support hinaus auch zwei E-Mail-Adressen in den FAQ der Dokumentation, wobei eine für Support Fragen und die zweite eher für kommerzielle Anliegen und den Erwerb von Lizenzen vorgesehen ist.[115]

Genau wie *InfluxDB* bietet der Event Store *Event Store* außer der üblichen Dokumentation und Verweisen auf verschiedene Community-Seiten ebenfalls eine Auswahl von zwei kommerziellen Support Optionen an. Diese unterscheiden sich durch die offerierten Service-Zeiten, die Schnelligkeit der Rückmeldung, die persönliche Betreuung durch einen Account Manager und entsprechend auch durch die Kosten.[116] Somit kann beim Vergleich des gebotenen Supports für TSDBs und Event Stores festgestellt werden, dass teilweise nicht zu vernachlässigende Differenzen zwischen den einzelnen TSDBs bzw. Event Stores bestehen.

[112] Vgl. InfluxData (2019f).

[113] Vgl. OpenTSDB (2018b), KairosDB Team (2015b) und Wei/Merlino/Léauté/Lim (2019c).

[114] Vgl. The PostgreSQL Global Development Group (1996d-2019).

[115] Vgl. Timescale (2018b).

[116] Vgl. Event Store LLP (2018b).

4 Grundlagen des praktischen Vergleichs

4.1 Anwendungsfall

Neben dem im vorherigen Kapitel 3 vorgestellten theoretischen Vergleich von TSDBs und Event Stores wird in Kapitel 5 darüber hinaus eine ausgewählte TSDB mit einem ausgewählten Event Store anhand eines konkreten Anwendungsfalles verglichen. Im Folgenden werden zunächst die Daten beschrieben, die für den praktischen Vergleich der DBMS in Kapitel 5 verwendet werden. Anschließend werden unter Berücksichtigung der Hardware und des Betriebssystems, der Daten sowie weiteren Gegebenheiten des Anwendungsfalles Anforderungen definiert, die von den beiden DBMS, welche für den praktischen Vergleich ausgewählt werden, zu erfüllen sind.

4.1.1 Hardware und Betriebssystem

Für den praktischen Vergleich einer TSDB mit einem Event Store in Kapitel 5 steht ein Server im Hochschulnetzwerk der Technischen Hochschule Köln mit dem Namen lwivs32.gm.fh-koeln.de und der IP-Adresse 139.6.56.162 zur Verfügung. Als Betriebssystem wird Ubuntu 18.04.2 LTS verwendet. Der Server verfügt über zwei virtuelle CPU-Kerne und vier Gigabyte (GB) RAM bzw. Arbeitsspeicher. Als Festplattenspeicher (Hard Disk Drive (HDD)) sind 120 GB verfügbar. Zu beachten ist, dass auf dem Server noch weitere virtuelle Maschinen laufen – insgesamt über 20 – was die Performance ggf. beeinflussen kann.

4.1.2 Daten

Für den praktischen Vergleich einer ausgewählten TSDB mit einem ausgewählten Event Store in Kapitel 5 werden Daten der Apple Health App verwendet, die standardmäßig auf allen iPhone 4s oder neuer und iPod touch der fünften Generation oder neuer verfügbar ist. Die Apple Health App konsolidiert Gesundheitsdaten, die von den iPhone- bzw. iPod-Usern manuell eingegeben werden können, wie bspw. Größe und Gewicht, sowie Daten, die vom iPhone, der Apple Watch und von Apps von Drittanbietern automatisch erfasst werden und fasst sie in die vier Kategorien „Aktivität“, „Schlaf“, „Achtsamkeit“ und „Ernährung“ zusammen. Unter die Kategorie „Aktivität“ fallen bspw. die vom iPhone sowie der Apple Watch gezählten zurückgelegten Schritte, Distanzen und erklommenen Stockwerke. Eine Apple Watch misst zusätzlich zu diesen drei Aktivitäten bspw. auch den Kalorienverbrauch und die Herzfrequenz.[117] Da diese Aktivitätsdaten im Zeitverlauf erfasst und in der Apple Health App gespeichert werden, stel-

[117] Vgl. Apple (2019a).

len sie Zeitreihendaten dar und eignen sich somit als Daten, die im Rahmen des praktischen Vergleichs in den beiden DBMS gespeichert und abgefragt werden können.

Die in der Apple Health App gesammelten Daten können über das iPhone in einem ZIP-komprimierten Ordner exportiert und auf unterschiedliche Art und Weise geteilt und gespeichert werden; z.B. ist ein Versand des ZIP-komprimierten Ordners als E-Mail oder Nachricht sowie das Speichern in der Dropbox möglich. In dem ZIP-komprimierten Ordner mit der Standard-Bezeichnung „Export.zip“ befindet sich ein weiterer Ordner mit dem Namen „apple_health_export“, in dem zwei XML-Dateien enthalten sind, nämlich „Export_cda.xml“ und „Export.xml“. Die XML-Datei „Export_cda.xml“ enthält klinische Daten aus der Apple Health App im Format des Standards Clinical Document Architecture (CDA), welcher dem Austausch von klinischen Dokumenten zwischen Gesundheitsdienstleistern dient.[118] In der XML-Datei „Export.xml“ befinden sich wiederum alle manuell durch den iPhone- bzw. iPod-User und alle automatisch durch das iPhone, die Apple Watch oder Apps von Drittanbietern in der Apple Health App erfassten Daten und somit auch die Aktivitätsdaten, die für den praktischen Vergleich einer TSDB und eines Event Stores in Kapitel 5 verwendet werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE HealthData [
<!-- HealthKit Export Version: 8 -->
<!ELEMENT HealthData (ExportDate,Me,(Record|Correlation|Workout|ActivitySummary|ClinicalRecord)*)>
<!ATTLIST HealthData
  locale CDATA #REQUIRED
```

Abbildung 4-1: Anfang einer beispielhaften „Export.xml“-Datei

Zusammenfassend bestehen die „Export.xml“-Dateien aus einem „HealthData“-Element, welches wiederum aus einem „ExportDate“-Element, einem „Me“-Element und keinem oder mehreren Elementen der Typen „Record“, „Correlation“, „Workout“, „ActivitySummary“, „ClinicalRecord“ sowie dem Attribut „locale“ besteht, wie in der Abbildung 4-1 zu sehen ist. Die ersten vier Zeilen eines beispielhaften „HealthData“-Elements, welche für eine bessere Lesbarkeit auf mehrere Zeilen verteilt wurden, werden in der Abbildung 4-2 gezeigt. Das Attribut „locale“ gibt das Land an, welches im Apple App Store ausgewählt ist. „ExportDate“ zeigt den Zeitpunkt, zu dem die Daten aus der Apple Health App exportiert wurden. Hinter dem „Me“-Element befinden sich Daten des Typs „HKCharacteristicType“. Dieser Typ repräsentiert Daten, die sich im Zeitverlauf normalerweise nicht ändern, wie in der Abbildung 4-2 z.B. das Geburtsdatum, das Geschlecht, die Blutgruppe und der Hauttyp.[119] Diese Daten können manuell vom iPhone- bzw. iPod-User in der Apple Health App eingegeben werden.

[118] Vgl. HL7 Deutschland e.V. (o.J.).

[119] Vgl. Apple (2019b).

```
<HealthData locale="de_DE">
 <ExportDate value="2019-02-17 14:53:51 +0100"/>
 <Me HKCharacteristicTypeIdentifierDateOfBirth="HKDateOfBirthNotSet"
HKCharacteristicTypeIdentifierBiologicalSex="HKBiologicalSexNotSet"
HKCharacteristicTypeIdentifierBloodType="HKBloodTypeNotSet"
HKCharacteristicTypeIdentifierFitzpatrickSkinType="HKFitzpatrickSkinTypeNotSet"/>
 <Record type="HKQuantityTypeIdentifierStepCount"
         sourceName="Kathrin's iPhone"
         sourceVersion="10.1.1"
         device="&lt;&lt;HKDevice: 0x2815b7930&gt;, name:iPhone, manufacturer:Apple, model:iPhone,
hardware:iPhone9,3, software:10.1.1&gt;"
         unit="count"
         creationDate="2016-11-21 18:30:33 +0100"
         startDate="2016-11-21 15:02:45 +0100"
         endDate="2016-11-21 15:10:38 +0100"
         value="11"/>
```

Abbildung 4-2: Ausschnitt eines beispielhaften „HealthData“-Elements einer „Export.xml“-Datei

Die „Record“-Elemente bestehen jeweils aus einem „type“, welcher die erfassten Daten bzw. die Messung identifiziert, Informationen zur Herkunft der Daten („sourceName“, „sourceVersion“), Informationen zum Gerät, von welchem die Daten erfasst wurden („device“), die Einheit, in der die Daten erfasst sind („unit“), den Zeitpunkt, zu dem die Daten gespeichert wurden („creationDate“), den Zeitpunkt, zu dem die Messung der Daten begonnen und geendet hat („startDate“, „endDate“) und schließlich dem Wert der Messung („value“) (vgl. Abbildung 4-2). Für den praktischen Vergleich der DBMS werden lediglich Daten des „Record types“ „HKQuantityType“ verwendet, zumal diese Daten numerische Werte enthalten, die im Zeitverlauf erfasst werden und sich als Zeitreihendaten somit für die Speicherung und Abfrage in TSDBs und Event Stores eignen. Die übrigen Element-Typen des Datenexports aus der Apple Health App, „Correlation“, „Workout“, „ActivitySummary“, „ClinicalRecord“, werden im Rahmen des Vergleichs der beiden DBMS in Kapitel 5 nicht verwendet und daher nicht näher betrachtet.

Die Zeitabstände zwischen den Zeitstempeln der einzelnen Messwerte des „Record types“ „HKQuantityType“ sind dabei abhängig von der Intensität, mit der der User die jeweilige Aktivität ausführt. Geht ein User bspw. eine Stunde lang spazieren, werden für diesen einstündigen Zeitraum mehr Messwerte des „Record types“ „HKQuantityType“ „StepCount“ vorliegen als für einen einstündigen Zeitraum, in dem der User am Schreibtisch sitzt. Außerdem liegt es nahe, dass ein User in einem bestimmten Zeitraum mehr Schritte geht oder läuft als Stockwerke erklimmt, was zudem bedeutet, dass für den in den Apple Health Daten erfassten „Record type“ „HKQuantityType“ „StepCount“ insgesamt mehr Messwerte vorliegen, als für den „Record type“ „HKQuantityType“ „FlightsClimbed“. Die Aktivitätsdaten der Apple Health App stellen somit unregelmäßige Zeitreihendaten dar, deren Erfassung von Events abhängig ist, nämlich dass der User die jeweils gemessene Aktivität ausführt.

Insgesamt liegen für den praktischen Vergleich 27 „Export.xml“-Dateien aus der Apple Health App unterschiedlicher Apple iPhone-User vor. Sie sind jeweils in einem Ordner mit dem Namen „exportx“ auf dem zur Verfügung stehenden Server abgelegt, wobei „x“ eine laufende Nummer von eins bis 27 ist. Die 27 Ordner („export1“ bis „export27“) sind wiederum allesamt in einem Ordner mit der Bezeichnung „export“ gespeichert. Die 27 „Export.xml“-Dateien sind entzippt zwischen 684 Kilobyte (KB) und 248.246 KB groß. Die Größe der Exporte bzw. der darin enthaltenen Datenmengen ist zum einen vom

Zeitraum, in dem die Daten gemessen wurden und somit auch im Export vorliegen, und zum anderen von der Anzahl der unterschiedlichen „Record types“ bzw. Messungen abhängig. Während ein iPhone nur die drei zuvor genannten Aktivitäten Schritte („StepCount“), zurückgelegte Distanz („DistanceWalkingRunning“) und Stockwerke („FlightsClimbed“) erfasst, können die Exporte darüber hinaus auch weitere Messungen der Apple Watch, wie Herzfrequenz („HeartRate“), Kalorienverbrauch bei physikalischer Aktivität („ActiveEnergyBurned“) und die beim Fahrradfahren zurückgelegte Distanz („DistanceCycling“) oder Messungen aus Apps von Drittanbietern beinhalten. Dies ist der Fall, wenn der iPhone-User zusätzlich eine Apple Watch besitzt bzw. Apps von Drittanbietern nutzt, deren Daten ebenfalls in der Apple Health App erfasst werden. Außerdem können auch Messungen von unterschiedlichen iPhones, iPods oder Apple Watches in einer „Export.xml“-Datei enthalten sein, falls der User mehrere unterschiedliche Geräte mit der gleichen Apple-ID zur Identifikation und Synchronisation nutzt.

Von den insgesamt 27 „Export.xml“-Dateien, die für den praktischen Vergleich einer TSDB mit einem Event Store vorliegen, beinhalten neun „Export.xml“-Dateien neben den vom iPhone erfassten Aktivitätsdaten auch Aktivitätsdaten einer Apple Watch. Da die Daten von Apps anderer Anbieter nur in wenigen Einzelfällen der 27 „Export.xml“-Dateien vorliegen, werden für den praktischen Vergleich nur die vom iPhone und der Apple Watch gemessenen Daten betrachtet.

Messung	Erklärung	Einheit	Häufigkeit
ActiveEnergyBurned	Kalorien, die der User sich ausruhend verbrannt hat	Kilokalorie	9
AppleExerciseTime	Zeit, die sich der User mindestens mit der Intensität eines schnellen Spaziergangs oder einer stärkeren Intensität bewegt hat	Minute	9
BasalEnergyBurned	Kalorien, die der User aktiv verbrannt hat	Kilokalorie	9
DistanceCycling	Distanz, die der User Fahrrad fahrend zurückgelegt hat	Kilometer	1
DistanceWalkingRunning	Distanz, die der User gehend oder laufend zurückgelegt hat	Kilometer	27
FlightsClimbed	Anzahl der Treppenläufe, die der User hinaufgestiegen ist	Anzahl	27
HeartRate	Herzfrequenz des Users	Anzahl/Minute	9
HeartRateVariabilitySDNN	Standardabweichung des Zeitabstands zwischen zwei Herzschlägen	Millisekunde	9
StepCount	Anzahl der Schritte, die der User zurückgelegt hat	Anzahl	27

Tabelle 4-1: Messungen des „Record types“ „HKQuantityType“ aus den 27 „Export.xml“-Dateien der Apple Health App[120]

[120] Vgl. Apple (2019b).

In der Tabelle 4-1 ist eine Übersicht der unterschiedlichen Messungen aus den für den praktischen Vergleich zur Verfügung stehenden Apple Health Daten zu sehen. Neben einer Beschreibung der Messungen gibt die Tabelle 4-1 auch Auskunft darüber, in wie vielen der 27 „Export.xml“-Dateien die einzelnen Messungen vorkommen (Häufigkeit).

4.1.3 Anforderungen an das Datenbank Management System

Einzelne TSDBs und Event Stores unterscheiden sich in Bezug auf funktionale sowie nicht-funktionale Kriterien (vgl. Kapitel 3), sodass im konkreten Anwendungsfall zu entscheiden ist, welche TSDB bzw. welcher Event Store für die jeweils geltenden Anforderungen am besten geeignet ist. Aus der für den praktischen Vergleich zur Verfügung stehenden Hardware (vgl. Kapitel 4.1.1) ergibt sich zunächst eine übergeordnete Anforderung an die zu verwendende TSDB bzw. den Event Store, welche lautet:

(1) Die TSDB bzw. der Event Store sollte auf dem Server mit dem Betriebssystem Ubuntu 18.04.2 LTS, zwei virtuellen CPU-Kernen, vier GB RAM und 120 GB HDD-Speicher installiert werden können.

Erfüllt die TSDB bzw. der Event Store diese erste Anforderung nicht, müsste der für den praktischen Vergleich zur Verfügung stehenden virtuellen Maschine eine höhere Anzahl von CPU-Kernen und/oder mehr GB RAM zugeteilt werden, was zusätzlichen Aufwand für den Administrator des Servers bzw. für die CampusIT der Technischen Hochschule Köln bedeuten würde und daher, wenn möglich, verhindert werden soll.

Wie in Kapitel 3.1.2 beschrieben, können die Daten sowohl bei TSDBs als auch bei Event Stores in unterschiedlichen Systemen, nämlich bereits bestehenden RDBMS, NoSQL DBMS oder unabhängigen Systemen der TSDB bzw. des Event Stores, gespeichert werden. Zumal der Fokus auf dem Vergleich der beiden DBMS Arten TSDB und Event Store liegt, ergibt sich in diesem bestimmten Anwendungsfall folgende zweite, funktionale Anforderung:

(2) Die TSDB bzw. der Event Store sollte über ein eigens für die TSDB bzw. den Event Store konzipiertes Speichersystem verfügen.

Die im vorherigen Kapitel 4.1.2 beschriebenen Daten, die für den praktischen Vergleich im folgenden Kapitel 5 verwendet werden, stellen Zeitreihendaten dar, die in unregelmäßigen Zeitabständen in der Apple Health App gespeichert werden. Diese Unregelmäßigkeit der von der Apple Health App erfassten Zeitreihendaten stellt eine Herausforderung an das DBMS dar, in dem die Daten gespeichert und abgefragt werden sollen, zumal bei der Berechnung von Mittelwerten einer Messung zusätzlich die Zeitabstände zwischen den einzelnen Zeitstempeln der Messwerte berücksichtigt werden

müssen.[121] Aus dieser Besonderheit der Daten wird folgende dritte, funktionale Anforderung an die TSDB bzw. den Event Store abgeleitet:

(3) Die TSDB bzw. der Event Store sollte für die Speicherung und Abfrage von unregelmäßigen Zeitreihendaten optimiert sein.

Inhaltlich stellen die Aktivitätsdaten aus der Apple Health App Gesundheitsdaten dar, deren Auswertung über einen längeren Zeitraum wichtige Erkenntnisse über den jeweiligen User ermöglichen kann. Ältere Daten sollten aufgrund von Speicherplatzengpässen somit nicht vollständig aus dem DBMS entfernt werden müssen, sondern langfristig in dem für die Speicherung und Abfrage verwendeten DBMS zur Verfügung stehen, da sonst Informationseinbußen entstehen könnten, woraus die vierte, ebenfalls funktionale Anforderung folgt:

(4) Mit der TSDB bzw. dem Event Store sollte eine Langzeit-Speicherung der Daten möglich sein.

Zumal die Datenmenge bei Zeitreihendaten schnell ansteigt und es für ein DBMS eine Herausforderung sein kann, diese großen Datenmengen über einen längeren Zeitraum zu speichern, ist es notwendig, dass das DBMS Lösungen anbietet, welche die langfristige Speicherung der Daten trotz der großen Datenmenge ermöglichen. Als Beispiellösung könnten Daten, die ein definiertes Alter überschritten haben, nur noch als aggregierte Werte im DBMS gespeichert werden, wie bspw. in Form eines Mittelwertes je Minute anstelle von 60 einzelnen Werten je Sekunde, während neuere Daten weiterhin auf einer detaillierteren Ebene zur Verfügung stehen.[122]

Während die funktionalen Anforderungen (Anforderungen zwei bis vier) hauptsächlich die Datenspeicherung betreffen, bezieht sich die folgende fünfte Anforderung an die TSDB bzw. den Event Store auf die Abfrage der Daten.

(5) Die Abfrage der Daten in der TSDB bzw. dem Event Store sollte mit SQL oder einer SQL-ähnlichen Abfragesprache erfolgen.

Diese Anforderung ist auf der weiten Verbreitung von SQL als Datenbanksprache begründet und soll zum einen den Vorteil hervorbringen, dass die die Daten abfragenden Anwender und Anwenderinnen der DBMS im Optimalfall auf ihr Wissen zu SQL bzw. auf ihre Erfahrungen mit dem Umgang mit SQL zurückgreifen können. Zum anderen sollen aggregierte Daten einfach abgefragt werden können.

Neben der übergeordneten ersten und den folgenden vier funktionalen Anforderungen ist auch eine nicht-funktionale Anforderung an die für den Vergleich auszuwählende TSDB bzw. den auszuwählenden Event Store zu stellen, welche den Support betrifft. Zumal in dem für den Vergleich vorliegendem Anwendungsfall in der Vergangenheit

[121] Vgl. Crowley (2018).

[122] Vgl. Bader (2016), S. 38.

weder Erfahrung mit einer TSDB noch mit einem Event Store gesammelt wurde, muss bei der Nutzung der beiden DBMS auf die veröffentlichte Dokumentation als Informationsquelle zurückgegriffen werden. Diese sollte ausführliche Informationen zu Themen wie der Installation des DBMS, über das Schreiben von Daten in das DBMS bis hin zur Abfrage der Daten enthalten. Hieraus entsteht folgende sechste Anforderung:

(6) Für die TSDB bzw. den Event Store sollte eine vollumfängliche Dokumentation zur Verfügung stehen.

Die folgende Tabelle 4-2 listet die sechs definierten Anforderungen auf, die bei der Auswahl einer TSDB und eines Event Stores für den Vergleich in Kapitel 5 zu berücksichtigen sind. Zusätzlich wird in der Tabelle 4-2 angegeben, welche Priorität der jeweiligen Anforderungen im vorliegenden Anwendungsfall zuzuschreiben ist.

Nr.	Anforderung	Priorität
(1)	Die TSDB bzw. der Event Store muss auf dem Server mit dem Betriebssystem Ubuntu 18.04.2 LTS, zwei virtuellen CPU-Kernen, vier GB RAM und 120 GB HDD-Speicher installiert werden können.	sehr hoch
(2)	Die TSDB bzw. der Event Store sollte über ein eigens für die TSDB bzw. den Event Store konzipiertes Speichersystem verfügen.	hoch
(3)	Die TSDB bzw. der Event Store sollte für die Speicherung und Abfrage von unregelmäßigen Zeitreihendaten optimiert sein.	hoch
(4)	Mit der TSDB bzw. dem Event Store sollte eine Langzeit-Speicherung der Daten möglich sein.	mittel
(5)	Die Abfrage der Daten in der TSDB bzw. dem Event Store sollte mit SQL oder einer SQL-ähnlichen Abfragesprache erfolgen.	niedrig
(6)	Für die TSDB bzw. den Event Store sollte eine vollumfängliche Dokumentation zur Verfügung stehen.	mittel

Tabelle 4-2: Anforderungen an die für den Vergleich auszuwählende TSDB bzw. den auszuwählenden Event Store mit ihrer Priorität

4.2 Auswahl einer Time Series Database und eines Event Stores für den praktischen Vergleich

Im Folgenden wird geprüft, inwieweit die in Kapitel 3 vorgestellten sechs TSDBs, *OpenTSDB*, *KairosDB*, *InfluxDB*, *Druid*, *PostgreSQL* und *TimescaleDB*, der vorgestellte Event Store, *Event Store*, sowie zwei weitere Event Stores, *IBM Db2 Event Store* und *NEventStore*, die im vorherigen Kapitel 4.1.3 definierten Anforderungen erfüllen. Neben der Tatsache, ob eine definierte Anforderung von einer TSDB bzw. von einem Event Store erfüllt wird oder nicht, wird im Auswahlprozess ebenfalls die Priorität der Anforderung berücksichtigt. Als Ergebnis dieses Abgleichs der definierten Anforderungen mit den Restriktionen sowie den funktionalen und nicht-funktionalen Eigenschaften der TSDBs und Event Stores wird jeweils eine TSDB und ein Event Store ausgewählt, die bzw. der sich für die Speicherung und Abfrage der Apple Health Daten im konkreten Anwendungsfall des praktischen Vergleichs in Kapitel 5 am besten eignet.

4.2.1 Auswahl einer Time Series Database

Zur Beurteilung, ob die sechs in Kapitel 3 vorgestellten TSDBs auf einem Server mit dem Betriebssystem Ubuntu 18.04.2 LTS, zwei virtuellen CPU-Kernen, vier GB RAM und 120 GB HDD-Speicher installiert werden können und somit die erste Anforderung erfüllen, werden die in den Dokumentationen der TSDBs festgehaltenen Systemvoraussetzungen näher betrachtet. Für die Installation von *OpenTSDB* wird in Bezug auf die Hardware und das Betriebssystem lediglich ein Linux-System vorausgesetzt.[123] Währenddessen gibt *KairosDB* weder eine Anforderung an die Hardware noch an das Betriebssystem vor.[124] Somit wird die erste Anforderung sowohl von *OpenTSDB* als auch von *KairosDB* erfüllt. In der Dokumentation von *InfluxDB* werden abhängig von der Anzahl der zu schreibenden Fields pro Sekunde, der Anzahl der auszuführenden Abfragen pro Sekunde sowie der Anzahl von einzigartigen „series“ (Messungen mit den gleichen Tags) verschiedene Belastungsstufen mit unterschiedlichen Richtlinien an die Hardware angegeben. Der für den Vergleich vorliegende Anwendungsfall mit den zur Verfügung stehenden Daten der Apple Health App ist hierbei in eine niedrige Belastung einzustufen, wofür zur Installation von *InfluxDB* zwei bis vier CPU-Kerne und zwei bis vier GB RAM empfohlen werden.[125] In Bezug auf das Betriebssystem ist *InfluxDB* u.a. auch für Ubuntu verfügbar, sodass *InfluxDB* die erste Anforderung ebenfalls erfüllt.[126] Die TSDB *Druid* gibt als Voraussetzung in Bezug auf das Betriebssystem u.a. Linux und bzgl. der Hardware zwei virtuelle CPU-Kerne und acht GB RAM vor.[127] Die Voraussetzung von acht GB RAM ist doppelt so hoch wie der in der ersten Anforderung definierte Arbeitsspeicher von vier GB RAM, was bedeutet, dass *Druid* die erste Anforderung nicht vollständig erfüllt. Die verbleibenden beiden TSDBs *PostgreSQL* und *TimescaleDB* geben lediglich eine Anforderung an das Betriebssystem vor, welche lautet, dass im Falle von einem Ubuntu-Betriebssystems mindestens die Version 14.04. oder neuer vorliegen muss.[128] Beide TSDBs erfüllen somit erste, übergeordnete Anforderung.

Die zweite Anforderung, dass die TSDB über ein eigens für die TSDB konzipiertes Speichersystem verfügt, wird von den in Kapitel 3 vorgestellten TSDBs nur von *InfluxDB* und *Druid* erfüllt. Beide DBMS sind speziell für Daten mit Zeitstempel optimiert.[129] Die anderen vier vorgestellten TSDBs aus Kapitel 3 erfüllen diese zweite Anforderung nicht, zumal sie im Fall von *OpenTSDB* und *KairosDB* ein bestehendes

[123] Vgl. OpenTSDB (2019a).

[124] Vgl. KairosDB Team (2015a).

[125] Vgl. InfluxData (2019c).

[126] Vgl. Bang/Anderson (2019d).

[127] Vgl. Wei/Merlino/Léauté/Lim (2019a).

[128] Vgl. The PostgreSQL Global Development Group (1996b-2019), Timescale (2018c).

[129] Vgl. InfluxData (2019d), Druid (2019b).

NoSQL DBMS für die Speicherung der Daten benötigen oder wie *PostgreSQL* und *TimescaleDB* die Daten in einem RDBMS speichern (vgl. auch Kapitel 3.1.2).[130]

Ob die sechs vorgestellten TSDBs die dritte Anforderung erfüllen, welche vorsieht, dass die TSDB für die Speicherung und Abfrage unregelmäßiger Zeitreihendaten optimiert ist, ist nicht für alle eindeutig bewertbar. Über *InfluxDB* wird im *InfluxData* Blog sowie in einem *InfluxData* Whitepaper angegeben, dass diese TSDB sowohl für regelmäßige als auch unregelmäßige Zeitreihendaten optimiert ist. Dies ist dadurch begründet, dass *InfluxDB* unregelmäßige Zeitreihendaten zu regelmäßigen Zeitreihendaten umwandeln kann, indem einzelne Werte über beliebige Zeitabstände aggregiert werden. Im *InfluxData* Whitepaper wird dies als klare Differenzierung zu anderen DBMS, u.a. der TSDB *OpenTSDB*, herausgestellt.[131] Somit kann festgehalten werden, dass *InfluxDB* die dritte Anforderung erfüllt, während sie für *OpenTSDB* nicht zutrifft. Da für die anderen TSDBs keine Informationen vorliegen, ob sie für unregelmäßige Zeitreihendaten optimiert sind oder nicht, wird diese Anforderung für diese TSDBs, nämlich *KairosDB*, *Druid*, *PostgreSQL* und *TimescaleDB*, nicht bewertet.

Eine Langzeit-Speicherung der Daten, wie sie die vierte Anforderung vorsieht, ist mithilfe der von den TSDBs angebotenen Lösungen lediglich bei *InfluxDB* sowie *Druid* und nur bedingt bei *KairosDB* und *TimescaleDB* möglich. Während *OpenTSDB* darauf ausgelegt ist, Originaldaten in ihrem jeweiligen Detaillierungsgrad zu speichern, sofern genügend Speicherplatz zur Verfügung steht, berechnet und speichert *OpenTSDB* selbst keine aggregierten Daten, sodass die vierte Anforderung nicht erfüllt wird.[132] *KairosDB* bietet wiederum sogenannte „roll-ups“ an, bei denen Abfragen geplant und kontinuierlich auf den in der TSDB gespeicherten Daten durchgeführt werden können. Bei diesen „roll-ups“ werden die Daten auf eine gröbere Zeitspanne aggregiert und die Ergebnisse zusätzlich als neue Metrik gespeichert. Zwar bietet diese Aggregation den Vorteil, dass die Abfrageperformance bei einer großen Menge von Daten verbessert wird, allerdings bietet *KairosDB* keine Möglichkeit, ältere Daten automatisch zu löschen, um die gespeicherte Datenmenge nicht stetig zu steigern.[133] Die vierte Anforderung wird von KariosDB somit nur bedingt erfüllt.

InfluxDB gewährleistet die Langzeit-Speicherung der Daten durch das sogenannte Downsampling, was bedeutet, dass die Daten nur für eine limitierte Zeit auf ihrer höchsten Detaillierungsebene in der TSDB gespeichert werden und die langfristige Datenspeicherung auf einer aggregierten Ebene mit geringerer Genauigkeit erfolgt. Um diesen Prozess des Downsamplings zu automatisieren, bietet *InfluxDB* zwei Features an, nämlich „Continuous Queries“ und „Retention Policies“. Eine „Continuous Query“ ist

[130] Vgl. The OpenTSDB Authors (2010a-2019), KairosDB (2015), Bader (2016), S. 35, Timescale (2019a).

[131] Vgl. Crowley (2018), Dix (2016), S. 3.

[132] Vgl. OpenTSDB (2019c).

[133] Vgl. KairosDB Team (2015e).

dabei eine Abfrage, welche in regelmäßigen Zeitabständen automatisch durchgeführt wird und ein „GROUP BY“-Statement mit einem Zeitintervall beinhalten muss, sodass die Daten für das definierte Zeitintervall aggregiert werden. Mithilfe der „Retention Policy“ (deutsch Aufbewahrungsrichtlinie) kann wiederum die Dauer („duration“) festgelegt werden, für welche die Daten in der *InfluxDB* gespeichert werden sollen. *InfluxDB* vergleicht hierzu den Zeitstempel des lokalen Servers mit dem Zeitstempel der Zeitreihendaten und löscht jegliche Daten, die älter als die Dauer der „Retention Policy“ sind.[134]

Um eine Langzeit-Speicherung bei *Druid* zu ermöglichen, ist eine Kombination verschiedener gebotenen Funktionen notwendig. Zum einen wird durch Apache *Druid* ein „roll-up“-Prozess angeboten, bei dem Daten bei der Aufnahme in das DBMS aggregiert werden, sodass die Menge der zu speichernden Daten reduziert wird. Diese „roll-up”-Prozesse zur Aggregation der Daten sind jedoch lediglich bei der initialen Datenspeicherung und nicht bei den bereits gespeicherten Daten möglich.[135] Zum anderen können, ähnlich wie bei *InfluxDB*, Regeln zur Aufbewahrung der Daten, sogenannte „Drop Rules“, genutzt werden, um Daten, die älter als der in der „Drop Rule“ definierte Zeitraum sind, zu löschen.[136] *Druid* erfüllt die vierte Anforderungen somit mit leichten Einschränkungen.

Während *PostgreSQL* keine Lösungen für die Langzeit-Speicherung der Zeitreihendaten zur Verfügung stellt[137], bietet *TimescaleDB* als Erweiterung von *PostgreSQL*, ähnlich wie die *InfluxDB*, auch einen Befehl an („drop_chunks()“), um Daten, die älter als eine festgelegte Zeitspanne sind, aus dem DBMS zu löschen. Eine Möglichkeit, diesen Befehl automatisch durchzuführen, besteht seit dem neuesten Release (Timescale 1.2) in Form einer „Data Retention Policy“, welche allerdings nur bei der kostenpflichtigen Enterprise Edition enthalten ist.[138] Weitere Lösungen, die eine Langzeit-Speicherung der Daten ermöglichen, wie kontinuierliche Aggregationen oder „roll-ups“, sind zwar im Rahmen von Timescale 1.2 angekündigt, zum Zeitpunkt der Auswahl einer TSDB für den praktischen Vergleich in Kapitel 5 jedoch noch nicht verfügbar, sodass auch für *TimescaleDB* festgehalten wird, dass die vierte Anforderung nur bedingt erfüllt wird.[139]

Die fünfte Anforderung, dass die Abfrage der Daten in der TSDB mit SQL oder einer SQL-ähnlichen Abfragesprache erfolgen soll, wird von den TSDBs *PostgreSQL* und *TimescaleDB* uneingeschränkt erfüllt, was auf deren Datenspeicherung in einem RDBMS zurückzuführen ist.[140] Im Gegensatz dazu unterstützen *OpenTSDB* und

[134] Vgl. Bang (2019a).

[135] Vgl. Wei/Lim (2019).

[136] Vgl. Merlino u. a. (2019).

[137] Vgl. Druid (2019b)

[138] Vgl. Timescale (2018a).

[139] Vgl. Freedman (2019).

[140] Vgl. Timescale (2019a), The PostgreSQL Global Development Group (1996a-2019).

KairosDB kein SQL als Abfragesprache.[141] In der Dokumentation von *OpenTSDB* wird sogar darauf hingewiesen, dass die Abfrage „etwas knifflig“ sein kann.[142] Dies ist genau gegenteilig zur Intention der definierten Anforderung an die TSDBs, SQL als Abfragesprache zu unterstützen, zumal damit eine möglichst einfache Abfrage der Daten ermöglicht werden soll. Abfragen von Daten in der *InfluxDB* können mit InfluxQL erfolgen, was einer SQL-ähnlichen Abfragesprache entspricht und somit die fünfte Anforderung erfüllt.[143] In der TSDB *Druid* werden Abfragen nativ in Form von JSON ausgedrückt und erfolgen über eine HTTP REST Abfrageoberfläche.[144] Allerdings steht darüber hinaus auch Apache *Druid* SQL, eine integrierte SQL-Zugriffsschicht, zur Verfügung, die auf Apache Calcite beruht und SQL in die native Abfragesprache von *Druid* übersetzt. Abgesehen von einem geringfügigen Aufwand zur Übersetzung der SQL-Abfrage sind dabei keine weiteren Performanceeinbußen gegenüber der nativen Abfragesprache von *Druid* zu erwarten.[145] Somit können auch bei der TSDB *Druid* Daten mit SQL abgefragt werden, was die fünfte Anforderung erfüllt.

Die sechste, nicht-funktionale Anforderung, dass für die TSDB eine vollumfängliche Dokumentation zur Verfügung stehen sollte, wird als einzige Anforderung von allen sechs betrachteten TSDBs erfüllt. Die Dokumentationen beinhalten allesamt Erklärungen zur Installation und Konfiguration der jeweiligen TSDB, zum Schreiben von Daten in die TSDB sowie zum Abfragen der in der TSDB gespeicherten Daten. Während die Dokumentationen von *OpenTSDB*, *KairosDB*, *InfluxDB*, *Druid* und *TimescaleDB* jeweils direkt auf der Webseite der einzelnen TSDBs integriert und abrufbar sind, ist die Dokumentation von *PostgreSQL* als PDF Dokument verfügbar.[146]

In der folgenden Tabelle 4-3 ist eine Zusammenfassung des Abgleichs der Anforderungen mit den jeweiligen Restriktionen und funktionalen sowie nicht-funktionalen Eigenschaften der TSDB zu finden. Sie zeigt, dass nur eine TSDB, nämlich *InfluxDB*, alle fünf definierten Anforderungen uneingeschränkt erfüllt. Aufgrund dieser Eindeutigkeit ist keine zusätzliche Berücksichtigung der Priorität der einzelnen Anforderung bei der Entscheidungsfindung notwendig und *InfluxDB* wird folglich als TSDB für den praktischen Vergleich in Kapitel 5 ausgewählt.

[141] Vgl. OpenTSDB (2019b), KairosDB Team (2015c), KairosDB Team (2015d).

[142] Vgl. OpenTSDB (2019b).

[143] Vgl. Bang (2018a).

[144] Vgl. Wei/Merlino/Léauté/Lim (2019b).

[145] Vgl. Wei/Merlino/Léauté/Allen (2019).

[146] Vgl. Vgl. OpenTSDB (2018b), KairosDB Team (2015b), Bang (2018b), Wei/Merlino/Léauté/Lim (2019c), Timescale (2018d) und The PostgreSQL Global Development Group (1996c-2019).

Priorität	OpenTSDB	KairosDB	InfluxDB	Druid	Post-greSQL	Timesca-leDB
Anforderung (1)	Die TSDB bzw. der Event Store muss auf dem Server mit dem Betriebssystem Ubuntu 18.04.2 LTS, zwei virtuellen CPU-Kernen, vier GB RAM und 120 GB HDD-Speicher installiert werden können.					
sehr hoch	trifft zu	trifft zu	trifft zu	trifft nicht zu	trifft zu	trifft zu
Anforderung (2)	Die TSDB bzw. der Event Store sollte über ein eigens für die TSDB bzw. den Event Store konzipiertes Speichersystem verfügen.					
hoch	trifft nicht zu	trifft nicht zu	trifft zu	trifft zu	trifft nicht zu	trifft nicht zu
Anforderung (3)	Die TSDB bzw. der Event Store sollte für die Speicherung und Abfrage von unregelmäßigen Zeitreihendaten optimiert sein.					
hoch	trifft nicht zu	nicht angegeben	trifft zu	nicht angegeben	nicht angegeben	nicht angegeben
Anforderung (4)	Mit der TSDB bzw. dem Event Store sollte eine Langzeit-Speicherung der Daten möglich sein.					
mittel	trifft nicht zu	trifft nur bedingt zu	trifft zu	trifft mit leichten Einschränkungen zu	nicht verfügbar	trifft nur bedingt zu
Anforderung (5)	Die Abfrage der Daten in der TSDB bzw. dem Event Store sollte mit SQL oder einer SQL-ähnlichen Abfragesprache erfolgen.					
niedrig	SQL wird nicht unterstützt	SQL wird nicht unterstützt	SQL-ähnliche Abfragesprache wird unterstützt	SQL wird durch Apache Calcite Adapter unterstützt	SQL als Abfragesprache wird unterstützt	SQL als Abfragesprache wird unterstützt
Anforderung (6)	Für die TSDB bzw. den Event Store sollte eine vollumfängliche Dokumentation zur Verfügung stehen.					
mittel	trifft zu	trifft zu	trifft zu	trifft zu	trifft zu	trifft zu

Tabelle 4-3: Abgleich der Anforderungen mit den Eigenschaften der TSDBs

4.2.2 Auswahl eines Event Stores

Bei der Auswahl eines Event Stores für den praktischen Vergleich in Kapitel 5 werden neben dem in Kapitel 3 vorgestellten Event Store *Event Store* zusätzlich auch die bekannten Event Stores *IBM Db2 Event Store* und *NEventStore* berücksichtigt.[147] *IBM Db2 Event Store* ist eine In-Memory-Datenbank[148], die darauf ausgelegt ist, Streaming-Daten in ereignisgesteuerten Anwendungen schnell aufzunehmen und zu analysieren.[149] Für diesen Event Store wird sowohl eine kostenfreie Version, die *IBM Db2*

[147] Vgl. solid IT gmbh (2019).

[148] Eine In-Memory-Datenbank ist ein DBMS, welches seine Daten im Haupt- bzw. Arbeitsspeicher (RAM) und nicht auf dem herkömmlichen Festplattenspeicher ablegt, wodurch höhere Zugriffsgeschwindigkeiten erreicht werden können (vgl. Luber/Litzel (2017a)).

[149] Vgl. IBM (o.J.b).

Event Store Developer Edition mit einigen Einschränkungen, als auch eine kostenpflichtige, lizenzierte Version, nämlich *IBM Db2 Event Store Enterprise Edition*, angeboten.[150] Da für den Vergleich in Kapitel 5 nur die kostenfreie *Developer Edition* in Frage kommt, wird auch nur diese im Auswahlprozess berücksichtigt. Der dritte zur Wahl stehende Event Store, *NEventStore*, wird als "persistence library" bezeichnet, die verwendet wird, um verschiedene Speicherimplementierungen im Kontext von Event Sourcing zu abstrahieren. Das Anwendungsfeld des *NEventStore*s liegt insbesondere auf CQRS und Domain-Driven-Design (vgl. auch Kapitel 2.2.3).[151]

Auch im Auswahlprozess eines Event Stores wird zunächst geprüft, ob die drei vorgestellten Event Stores die erste Anforderung, dass sie auf dem Server mit dem Betriebssystem Ubuntu 18.04.2 LTS, zwei virtuellen CPU-Kernen, vier GB RAM und 120 GB HDD-Speicher installiert werden können, erfüllen. Laut der Dokumentation von *Event Store* lässt sich dieser unter Linux installieren und benötigt keine Voraussetzungen bzgl. der Hardware, sodass die erste Anforderung von *Event Store* erfüllt wird.[152] Aufgrund der Tatsache, dass es sich beim *IBM Db2 Event Store* um eine In-Memory-Datenbank handelt, ergeben sich bestimmte Systemanforderungen an das System, auf dem der *IBM Db2 Event Store Developer Edition* installiert werden soll. Einerseits kann der *IBM Db2 Event Store* auf einem Linux-Betriebssystem installiert werden, was einen Teil der ersten Anforderung erfüllt. Andererseits wird in Bezug auf die Hardware ein Minimum von acht anstatt den in der ersten Anforderung definierten zwei CPU-Kernen und mindestens zwölf anstelle der in der ersten Anforderung festgelegten vier GB RAM Arbeitsspeicher vorausgesetzt, wodurch die erste Anforderung insgesamt nicht erfüllt wird.[153] Beim *NEventStore* handelt es sich um eine .NET-Anwendung, die mindestens das .NET Framework 4.5 oder neuer, welches nur unter Windows installiert werden kann, oder mindestens .NET Core 2.0 oder neuer, was ebenfalls unter Linux installiert werden kann, voraussetzt. Bzgl. der Hardware wird keine Voraussetzung angegeben, sodass *NEventStore* die erste Anforderung erfüllt.[154]

Der Event Store *Event Store* erfüllt ebenfalls die zweite Anforderung, zumal dieser, wie in Kapitel 3.1.2 beschrieben, kein bestehendes DBMS zur Speicherung der Daten, sondern ein eigens konzipiertes Speichersystem verwendet.[155] Auch der *IBM Db2 Event Store* speichert die Daten unabhängig von bereits bestehenden DBMS, sodass auch hier die in der zweiten Anforderung definierte Bedingung zutrifft.[156] Im Gegensatz dazu erfolgt die persistente Datenspeicherung beim *NEventStore* in einem anderen

[150] Vgl. IBM (o.J.e).

[151] Vgl. Giorgetti (2019c).

[152] Vgl. Chinchilla (2019a).

[153] Vgl. IBM (o.J.d).

[154] Vgl. Giorgetti (2019c), Microsoft (2019).

[155] Vgl. Event Store LLP (2018f).

[156] Vgl. IBM (o.J.b).

System.[157] Dabei kann aus einer Auswahl von RDBMS, u.a. *Microsoft SQL Server* und *PostgreSQL*, Cloud-basierten DBMS, bspw. *Microsoft SQL Azure*, und Dokumenten-orientierten NoSQL DBMS, wie *MongoDB*, gewählt werden.[158] Die zweite Anforderung wird vom *NEventStore* somit nicht erfüllt.

Anders als Time Series Daten, die üblicherweise in regelmäßigen Zeitabständen gemessen folglich auch gespeichert werden, treffen Events im Zeitverlauf i.d.R. nach unvorhersehbaren Mustern ein (vgl. Kapitel 3.1.1) und stellen somit unregelmäßige Zeitreihendaten dar.[159] Zumal Event Stores auf die Speicherung von solchen unregelmäßigen Events spezialisiert sind und Events, sofern sie im Zeitverlauf erfasst und gespeichert werden auch Zeitreihen darstellen, kann daraus abgeleitet werden, dass Event Stores auch für die Speicherung von unregelmäßigen Zeitreihendaten geeignet sind. Die dritte Anforderung wird somit von allen drei Event Stores erfüllt.

Die vierte Anforderung, dass der Event Store eine Langzeit-Speicherung der Daten ermöglichen soll, scheint aufgrund der „append-only“ Eigenschaft von Event Stores zunächst von jeglichen Event Stores erfüllt zu werden, da diese Eigenschaft bedeutet, dass Events lediglich hinzugefügt und bereits gespeicherten Events weder geändert noch gelöscht werden (vgl. Kapitel 2.2.1).[160] Jedoch wird im Rahmen der vierten Anforderung ebenfalls gefordert, dass das DBMS Lösungen anbietet, um mit der stetig steigenden Datenmenge umzugehen. Der Event Store *Event Store* bietet hierfür die Erstellung von rollierenden Snapshots an, sodass bei Abfragen nicht alle Events seit Beginn der Erfassung, sondern nur die zeitlich nach dem Snapshot liegenden Events verarbeitet werden müssen (vgl. ebenfalls Kapitel 2.2.1). Durch die regelmäßige Erstellung solcher rollierenden Snapshots kann die Performance bei Abfragen optimiert werden.[161] Somit ist beim *Event Store* eine langfristige Speicherung der Daten sowie eine Abfrage dieser trotz großer Datenmenge möglich und die vierte Anforderung erfüllt.

Beim *IBM Db2 Event Store* erfolgt die Langzeit-Speicherung der Daten in der sogenannten „shared zone“ nachdem die erfassten Daten in der „log zone“ verarbeitet wurden. Für die persistent in die „shared zone“ geschriebenen Daten können ebenfalls konsistente Snapshots erstellt werden.[162] Beim *NEventStore* können ebenso Snapshots erstellt werden, um die Performance zu verbessern. Dabei ist zu beachten, dass die Snapshots beim *NEventStore* abhängig vom gewählten Speichersystem unterschiedlich zu implementieren sind. Folglich bieten auch die Event Stores *IBM Db2*

[157] Vgl. Giorgetti (2019a).

[158] Vgl. Giorgetti (2019d).

[159] Vgl. Seidemann/Seeger (2017), S. 145; Shoshani (2009), S. 2995.

[160] Vgl. Rybicki (2018), S. 48.

[161] Vgl. Chinchilla (2018b).

[162] Vgl. IBM (o.J.c).

Event Store und *NEventStore* Lösungen bzgl. der langfristigen Speicherung von Daten an und erfüllen somit die vierte, funktionale Anforderung.

Die fünfte Anforderung, dass die Abfrage der Daten im Event Store mit SQL oder einer SQL-ähnlichen Abfragesprache erfolgen soll, wird von dem Event Store *Event Store* nicht erfüllt. Abfragen können hier u.a. über das Admin UI oder mit einem GET-Request über die HTTP API erfolgen.[163] Im Gegensatz dazu wird die fünfte Anforderung beim *IBM Db2 Event Store* erfüllt, da SQL-Abfragen mithilfe von Spark SQL möglich sind.[164] Auch beim *NEventStore* kann die Datenabfrage mit SQL durchgeführt werden, vorausgesetzt es wurde ein RDBMS als System für die persistente Datenspeicherung gewählt.[165]

Eine einfache Dokumentation steht für alle drei betrachteten Event Stores zur Verfügung.[166] Die Dokumentationen der Event Stores *Event Store* und *IBM Db2 Event Store Developer Edition* sind strukturiert aufgebaut und beinhalten Erklärungen zur Installation, zum Schreiben und zum Abfragen von Event Daten sowie zu verschiedenen APIs. Die sechste, nicht-funktionale Anforderung, welche die Verfügbarkeit einer vollumfänglichen Dokumentation vorsieht, wird bei diesen beiden Event Stores somit erfüllt. Die Dokumentation des Event Stores *NEventStore* ist hingegen weniger ausführlich und für einen Leser ohne Erfahrungen mit Event Stores schwerer verständlich. Aus diesem Grund wird die sechste Anforderung beim *NEventStore* als nicht erfüllt bewertet.

Die folgende Tabelle 4-4 zeigt eine Zusammenfassung des zuvor beschriebenen Abgleichs der Anforderungen mit den jeweiligen Restriktionen sowie funktionalen und nicht-funktionalen Eigenschaften der drei Event Stores. Da der Event Store *NEvent-Store* mit nur drei der sechs Anforderungen im Vergleich zu den anderen Event Stores die wenigsten Anforderungen ohne Einschränkungen erfüllt, ist er im konkreten Anwendungsfall am wenigsten geeignet. Zumal die Event Stores *Event Store* und *IBM Db2 Event Store* (*Developer Edition*) wiederum beide fünf der sechs definierten Anforderungen erfüllen, werden für die Auswahl eines Event Stores ebenfalls die definierten Prioritäten der einzelnen Anforderungen betrachtet. Während der Event Store *IBM Db2 Event Store* die erste und mit der Einstufung „sehr hoch“ die am höchsten priorisierte Anforderung nicht erfüllt, wird vom Event Store *Event Store* die dritte Anforderung nicht erfüllt, welche lediglich eine niedrige Priorität aufweist. Unter Berücksichtigung der Prioritäten ist folglich der Event Store *Event Store* im konkreten Anwendungsfall am besten geeignet und wird daher für den praktischen Vergleich in Kapitel 5 ausgewählt.

[163] Vgl. Chinchilla (2019b).

[164] Vgl. IBM (o.J.a).

[165] Vgl. Giorgetti (2019d).

[166] Vgl. Leech u. a. (2019), IBM (o.J.f), Giorgetti (2019b).

Priorität	Event Store	IBM Db2 Event Store	NEventStore
Anforderung (1)	Die TSDB bzw. der Event Store muss auf dem Server mit dem Betriebssystem Ubuntu 18.04.2 LTS, zwei virtuellen CPU-Kernen, vier GB RAM und 120 GB HDD-Speicher installiert werden können.		
sehr hoch	trifft zu	trifft nicht zu	trifft zu
Anforderung (2)	Die TSDB bzw. der Event Store sollte über ein eigens für die TSDB bzw. den Event Store konzipiertes Speichersystem verfügen.		
hoch	trifft zu	trifft zu	trifft nicht zu
Anforderung (3)	Die TSDB bzw. der Event Store sollte für die Speicherung und Abfrage von unregelmäßigen Zeitreihendaten optimiert sein.		
hoch	trifft zu	trifft zu	trifft zu
Anforderung (4)	Mit der TSDB bzw. dem Event Store sollte eine Langzeit-Speicherung der Daten möglich sein.		
mittel	trifft zu	trifft zu	trifft zu
Anforderung (5)	Die Abfrage der Daten in der TSDB bzw. dem Event Store sollte mit SQL oder einer SQL-ähnlichen Abfragesprache erfolgen.		
niedrig	SQL wird nicht unterstützt	SQL wird unterstützt	SQL wird bei RDBMS als Speichersystem unterstützt
Anforderung (6)	Für die TSDB bzw. den Event Store sollte eine vollumfängliche Dokumentation zur Verfügung stehen.		
mittel	trifft zu	trifft zu	trifft nicht zu

Tabelle 4-4: Abgleich der Anforderungen mit den Eigenschaften der Event Stores

4.3 Vorgehensmodell des Vergleichs

Im Folgenden wird zunächst auf bereits existierende Benchmarks für TSDBs und Event Stores näher eingegangen, die für den Vergleich dieser beiden Arten von DBMS verwendet werden können. Anschließend wird das Vorgehen beschrieben, wie die im vorherigen Kapitel 4.2 ausgewählte TSDB *InfluxDB* mit dem Event Store *Event Store* in Kapitel 5 verglichen werden.

4.3.1 Benchmarking von Time Series Databases und Event Stores

Zur Bewertung und für den Vergleich der Performance von DBMS werden i.d.R. sogenannte Benchmarks verwendet. Ein solcher Benchmark ist eine Software, welche Performance-Parameter in einer klar definierten und steuerbaren Umgebung misst.[167] Auf Basis des Ergebnisses eines Benchmarkings von DBMS kann schließlich das für einen gegebenen Anwendungsfall am besten geeignetste DBMS gewählt werden.

In der Literatur sind verschiedene Benchmarks für TSDBs zu finden. Unter anderem ist hier der *TSDBBench* von *Bader* zu nennen. Diesem Benchmark liegt der *Yahoo! Cloud*

[167] Vgl. Bader (2016), S. 21.

Serving Benchmark (*YCSB*) zugrunde. Der *YCSB* ist ein Framework, das aus einem Workload generierenden Client und einer Reihe von Standard-Workloads besteht, die unterschiedliche Performance-Aspekte abdecken. Die wichtige Besonderheit am YCSB Framework ist seine Erweiterbarkeit, da sowohl neue Workloads definiert als auch neue Cloud DBMS durch eine Anpassung des Clients einem Benchmark unterzogen werden können. Der Code zur Beteiligung ist open source.[168] Im Rahmen des *TSDB-Bench* wurde der *YCSB* zu *YCSB-TS* erweitert, um Workloads in TSDBs ausführen zu können. Als Teil dieser Erweiterung wurden bspw. die Abfragen, die für das Benchmarking verwendet werden, auf für TSDBs typische Abfragen angepasst. Außerdem wurden Clients für eine Auswahl an TSDBs ergänzt.[169] Eine detaillierte Beschreibung des Benchmarks *TSDBBench* sowie Benchmarking Ergebnisse von zehn TSDBs, u.a. *InfluxDB*, *Druid*, *OpenTSDB*, *PostgreSQL* und *KairosDB*, sind in der Arbeit von *Bader* zu finden.[170]

Darüber hinaus wird als weiteres Beispiel eines Benchmarking-Verfahrens für TSDBs das *IoTDB-Benchmark* Framework von *Liu* und *Yuan* aufgeführt, welches speziell für Szenarien mit TSDBs und IoT-Anwendungen entwickelt wurde. Neben der Performance der TSDBs wird bei diesem Benchmarking auch der Verbrauch der Systemressourcen gemessen. Zudem berücksichtigt das *IoTDB-Benchmark* Framework gezielt die Besonderheiten von Time Series Daten, wie bspw. die Tatsache, dass Zeitreihendaten sowohl in nach dem Zeitstempel geordneter als auch in ungeordneter Reihenfolge in eine TSDB geschrieben werden können. Das *IoTDB-Benchmark* Tool ist dabei modular aufgebaut, sodass es skalierbar und für neue Funktionalitäten sowie neue TSDBs erweiterbar ist. Für die genaue Architektur des *IoTDB-Benchmark* Tools sowie Ergebnisse eines Benchmarks der vier TSDBs *InfluxDB*, *TimescaleDB*, KariosDB und *OpenTSDB* wird auf die Arbeit von *Liu* und *Yuan* verwiesen.[171]

Außerdem ist der open source Code für das Benchmarking der TSDB *InfluxDB* gegen die TSDB *OpenTSDB* als auch die NoSQL DBMS *Elasticsearch*, *Cassandra* und *MongoDB* veröffentlicht.[172] Die Ergebnisse dieser Benchmarks sind auf der Webseite von *InfluxData* als technische Paper zugänglich.[173]

Im Gegensatz dazu sind für das Benchmarking von Event Stores keine veröffentlichen Frameworks zu finden.

[168] Vgl. Cooper u. a. (2010), S. 144.

[169] Vgl. Bader (2016), S. 65 f.

[170] Vgl. Bader (2016), S. 45 ff.

[171] Vgl. Liu/Yuan (2019), S. 1 ff.

[172] Vgl. InfluxData (2019e).

[173] Vgl. InfluxData (2019g).

4.3.2 Vorgehen bei dem praktischen Vergleich der Time Series Database und des Events Stores

Der praktische Vergleich der in Kapitel 4.2 ausgewählten TSDB *InfluxDB* und des ausgewählten Event Stores *Event Store* soll sowohl die Performance bewerten als auch qualitative bzw. weiche Aspekte beinhalten. Wie aus dem Kapitel 4.3.1 hervorgeht, kann für den Performance-Vergleich der beiden Arten von DBMS kein bereits existierendes Benchmarking Framework verwendet werden, da zwar mehrere Benchmarks für TSDBs, aber keine Benchmarks für Event Stores oder TSDBs und Event Stores zusammen in einem Benchmark veröffentlicht sind. In dem Sinne, wie ein Benchmark im vorangehenden Kapitel 4.3.1 definiert wird, wird auch im Rahmen des praktischen Vergleichs im folgenden Kapitel 5 kein neues Benchmarking Framework für das Benchmarking von TSDBs und Event Stores entwickelt. Vielmehr wird im folgenden Vergleich die Performance der beiden DBMS unter den Gegebenheiten des Anwendungsfalles (vgl. Kapitel 4.1) analysiert und verglichen. Als quantitative Vergleichsgröße wird hierfür die Kennzahl „Anzahl der geschriebenen Metriken bzw. Events pro Sekunde“ beim Schreibprozess der Daten in die TSDB *InfluxDB* bzw. den Event Store *Event Store* betrachtet. Damit verhindert wird, dass das Ergebnis der Performance-Analyse durch einmalige externe Einflüsse verzerrt und somit verfälscht wird, werden mehrere Schreibvorgänge mit unterschiedlichen Batch Größen, d.h. variierenden Mengen von Metriken bzw. Events, die auf einmal in das DBMS zu schreiben sind, durchgeführt und Durchschnitte gebildet. Dadurch kann ebenfalls getestet werden, ob die verwendete Batch Größe Einfluss auf die Anzahl der geschriebenen Metriken bzw. Events pro Sekunde hat.

Der Vergleich anhand qualitativer bzw. weicher Aspekte soll wiederum die Erfahrungen, die bei der Implementierung der beiden Arten von DBMS und bei der Durchführung von Abfragen in den beiden DBMS gemacht werden, berücksichtigen. Hierzu gehört zum einen der Aufwand, der mit der Implementierung und der Durchführung von Abfragen verbunden ist. Zum anderen wird auch die Komplexität und die Usability in diesen qualitativen Vergleich mit einbezogen.

5 Praktischer Vergleich einer Time Series Database und eines Event Stores

5.1 Implementierung

Im Folgenden wird beschrieben, wie die beiden für den praktischen Vergleich ausgewählten DBMS implementiert werden, worunter u.a. ihre Installation und Konfiguration fällt. Darüber hinaus wird zum einen erläutert, wie die in Kapitel 4.1.2 beschriebenen Daten aufbereitet, d.h. in ein „importierfähiges“ Format gebracht werden, um sie in der ausgewählten TSDB *InfluxDB* und im ausgewählten Event Store *Event Store* speichern zu können. Zum anderen wird geschildert, wie die aufbereiteten Daten in die DBMS geschrieben werden. Abschließend wird die Implementierung der *InfluxDB* und des *Event Store* in Bezug auf die Punkte Installation und Konfiguration, Datenaufbereitung sowie den Schreibvorgang sowohl quantitativ als auch qualitativ verglichen. Zusätzlich werden Besonderheiten der beiden DBMS herausgestellt.

Zumal die Apple Health Daten auch personenbezogene Daten beinhalten und die in den beiden DBMS zu speichernden Daten im Rahmen des praktischen Vergleichs jedoch nicht auf bestimmte Personen zurückgeführt werden können sollen, werden die Apple Health Daten als Teil einer allgemeinen Aufbereitung anonymisiert. Diese Datenaufbereitung erfolgt mithilfe eines Skriptes in der Programmiersprache *R* (basicdataPreparation.R), welches im Anhang A zu finden ist. Alle Schritte der Datenaufbereitung einer XML-Datei sind dabei in der individuell für den Anwendungsfall entwickelten Funktion „dataPreparation“ zusammengefasst.

Wie in Kapitel 4.1.2 bereits erläutert, werden für den Vergleich der beiden DBMS nur die Daten der „Record“-Elemente aus den XML-Dateien verwendet. Diese werden als Teil der Funktion „dataPreparation“ extrahiert und zunächst in einen Dataframe geschrieben. Darüber hinaus werden Messwerte ausgeschlossen, die keinen Eintrag in den Spalten „device“ und „unit“ aufweisen. Anschließend werden, u.a. bei den Datumsangaben „startDate“ und „endDate“, einige Veränderungen des Datenformates vorgenommen. Als weiterer Schritt der Datenaufbereitung wird die Bezeichnung der einzelnen Messungen (Spalte „measurement“) angepasst, in dem lediglich der Name der Aktivität, bspw. „StepCount“, ohne den in den Originaldaten enthaltenen Präfix „HKQuantityTypeIdentifier“, in den Dataframe geschrieben wird.

Für die Anonymisierung der Daten wird die laufende Nummer in den Namen der Ordner, in dem die „Export.xml“-Dateien gespeichert sind („Export1“ bis „Export27“, vgl. Kapitel 4.1.2), als ID für die User („userID“) der Apple Health App verwendet. Darüber hinaus wird aus den in den „Record“-Elementen enthaltenen Informationen zu „device“ lediglich der Name des Geräts, von welchem die Daten erfasst werden, nämlich iPhone oder Apple Watch, extrahiert und als neue Spalte „deviceAnonymous“ im Dataframe erfasst. Da ein User allerdings mehrere Geräte mit dem gleichen Namen, z.B. mehrere iPhones, besitzen kann, wird zudem für jedes unterschiedliche Gerät eine eindeutige „userDeviceSourceID“ hinzugefügt, die aus der „userID“, dem „deviceAnonymous“ und

einer eindeutigen ID (beginnend bei eins) je unterschiedlichem „sourceName“ aus den „Record“-Elementen eines Users besteht. Besitzt der User mit der „userID“ „User1“ bspw. eine Apple Watch mit dem „sourceName“ „Apple Watch von User1“ und zwei iPhones, eins mit dem „sourceName“ „iPhone von User1“ und das zweite mit dem „sourceName“ „User1’s iPhone“, gibt es für diesen „User1“ die drei unterschiedlichen „userDeviceSourceIDs“ „User1_Apple Watch_1“, „User1_iPhone_2“ und „User1_iPhone_3“. Da die Daten mithilfe dieser neu definierten IDs eindeutig einem bestimmten Gerät eines Users zugeordnet werden können, sind die in den Originaldaten enthalten Informationen zu „sourceName“, „sourceVersion“ und „device“ in den Daten für den praktischen Vergleich nicht mehr notwendig. Folglich werden nur folgende Spalten des Dataframes für das Schreiben in die DBMS ausgewählt, die allesamt keine personenbezogenen Daten mehr enthalten: „unit“, „startDate“, „endDate“, „value“, „userID“, „deviceAnonymous“, „userDeviceSourceID“ und „measurement“.

5.1.1 InfluxDB

Für die Durchführung des praktischen Vergleichs wird die TSDB *InfluxDB* Version 1.7 verwendet. Es wurde nicht die neuere Version 2.0 gewählt, da es sich bei der Version 2.0 zum Zeitpunkt der Durchführung des Vergleichs noch um eine Alpha-Version handelte. Der Download und die Installation der *InfluxDB* 1.7 erfolgte gemäß der Anleitung für Ubuntu in der Dokumentation.[174] Die Konfiguration von *InfluxDB* geschieht durch die Konfigurationsdatei „InfluxDB.conf“. Zum einen können hier HTTP Endpunkte konfiguriert werden. Da der in der Konfigurationsdatei festgelegte Default-Port für den HTTP Service der Port mit der Nummer 8086 ist und dieser auf dem Server, auf dem die *InfluxDB* installiert ist, zum Zeitpunkt der Konfiguration schon vergeben ist, wird in der Konfigurationsdatei der Port mit der Nummer 8082 für den HTTP Service definiert. Zum anderen beinhaltet die Konfigurationsdatei auch Einstellungen zu sogenannten „Retention Policies“ (vgl. Kapitel 4.2.1). Aufgrund der Tatsache, dass im Rahmen des praktischen Vergleichs keine „Retention Policies“ definiert werden, werden diese in der Konfigurationsdatei deaktiviert. Eine weitere Einstellung, die in der Konfigurationsdatei angepasst wird, ist die „max-body-size“, welche die Größe eines Client Request Bodies in Bytes angibt.[175] Um zu verhindern, dass die von HTTP Clients gesendeten Daten das definierte Maximum überschreiten und die Requests aus diesem Grund fehlschlagen, wird das Limit aufgehoben, indem der Wert auf „0“ gesetzt wird. Des Weiteren werden die in der Konfigurationsdatei als Default festgelegten Werte für die „CacheMaxMemorySize“ und die „CacheSnapshotMemorySize“ von 1 GB und 25 Megabyte (MB) auf 500 MB und 20 MB reduziert. Für alle weiteren Einstellungen in der Konfigurationsdatei werden die Default-Werte verwendet.

[174] Vgl. Bang (2018b).

[175] Vgl. Bang/Anderson (2019a).

Für das Schreiben von Daten in die *InfluxDB* wird *Telegraf* Version 1.11 verwendet. *Telegraf* ist ein Plug-in getriebener Agent, mit dem sowohl Metriken und Events von verschiedenen Input-Quellen gesammelt bzw. erfasst als auch Daten an unterschiedliche Outputs gesendet werden können. Genau wie die *InfluxDB* gehört er zur *InfluxData* Plattform.[176] Auch bei dem Download und der Installation von *Telegraf* 1.11 diente die Beschreibung in der Dokumentation als Anleitung.[177] In der Konfigurationsdatei von *Telegraf* können neben Einstellungen zum *Telegraf* Agenten allgemein auch die gewünschten Inputs, d.h. von wo die Daten gesammelt werden sollen, und Outputs, d.h. wohin die gesammelten Daten gesendet werden sollen, konfiguriert werden. Dabei ist aus einer Vielzahl von bestehenden Input Plug-ins zu wählen, wie Plug-ins für verschiedene DBMS, andere Systeme, bspw. Cloud Plattformen, oder (IoT-)Sensoren.[178] Um die Apple Health Daten in die *InfluxDB* zu schreiben, wird das *Telegraf* Input Plug-in „File“ verwendet. Damit ist es möglich, Daten unterschiedlicher Dateiformate zu parsen und als Metrik in die *InfluxDB* zu schreiben.[179] Für das Schreiben der Daten in die *InfluxDB* wird im Rahmen des praktischen Vergleichs CSV als Dateiformat ausgewählt.

Damit die Daten als Ergebnis der Datenaufbereitung in einem in die *InfluxDB* „importierfähigen“ CSV-Format vorliegen, ist es notwendig, die zuvor beschriebene Funktion zur allgemeinen Datenaufbereitung („dataPreparation“) etwas anzupassen. Die angepasste Funktion der Datenaufbereitung wird in „InfluxDBdataPreparation“ umbenannt und in der Datei „InfluxDB-dataPreparation.R“ (s. Anhang B) gespeichert. Als Teil der Anpassung muss zum einen der Wert des „endDate“s, d.h. der Zeitpunkt, zu dem die Aktivität geendet ist, in ein Zeitformat der Go „reference time“ umgewandelt werden, sodass dieser Wert in der *InfluxDB* als Zeitstempel (timestamp) verwendet werden kann. Diese Umwandlung des Datentyps wird in der Funktion zur Datenaufbereitung ergänzt und der neue Zeitstempel in die Spalte „time“ im Dataframe geschrieben. Diese neu definierte Spalte „time“ wird anstelle der Spalte „endDate“ des Dataframes für das Schreiben in die DBMS ausgewählt. Um eine Konsistenz bei den Datentypen der Zeitangaben zu gewährleisten, wird das „startDate“ ebenfalls in das gleiche Zeitformat der Go „reference time“ wie der Zeitstempel „time“ gebracht. Zum anderen wird die Funktion zur Datenaufbereitung für die *InfluxDB* um die Speicherung des Dataframes als CSV-Datei in dem Ordner „cleanCSV“ ergänzt.

Neben dem *R*-Skript „InfluxDB-dataPreparation.R“ wird ein zweites Skript („InfluxDB-loop-dataPreparation.R“, vgl. Anhang C) verwendet, mit dem die Funktion „InfluxDB-dataPreparation“ aufgerufen wird und auf alle „Export.xml“-Dateien im Ordner „export“ auf dem Server angewendet wird.

[176] Vgl. InfluxData (2019h).

[177] Vgl. Bang (2019d).

[178] Vgl. Bang/Schmitz (2019).

[179] Vgl. Bang (2019e).

Als Ergebnis dieses Prozesses der Datenvorverarbeitung mit den beiden *R*-Skripten liegen somit 27 CSV-Dateien mit den Namen „User1“ bis „User27“ vor, die mithilfe des *Telegraf* Agenten in die *InfluxDB* geschrieben werden können. Dafür ist zum einen die *InfluxDB* als Output Plug-in und zum anderen „File“ als Input Plug-in in der „Telegraf.conf“-Datei zu konfigurieren. Für die Konfiguration des Outputs wird im Plug-in für die *InfluxDB* ([[outputs.InfluxDB]]) zunächst die URL definiert, an die die Daten gesendet werden sollen. Da für die auf dem Server installierte *InfluxDB* der Port mit der Nummer 8082 für den HTTP Service konfiguriert wurde, wird folgende URL im Output Plug-in definiert: http://127.0.0.1:8082, wie in der folgenden Abbildung 5-1 zu sehen ist.

```
# Configuration for sending metrics to InfluxDB
 [[outputs.influxdb]]
  ## The full HTTP or UDP URL for your InfluxDB instance.
  ##
  ## Multiple URLs can be specified for a single cluster, only ONE of the
  ## urls will be written to each interval.
  # urls = ["unix:///var/run/influxdb.sock"]
  # urls = ["udp://127.0.0.1:8080"]
   urls = ["http://127.0.0.1:8082"]

  ## The target database for metrics; will be created as needed.
  ## For UDP url endpoint database needs to be configured on server side.
   database = "thesisKS"

  ## The value of this tag will be used to determine the database.  If this
  ## tag is not set the 'database' option is used as the default.
  # database_tag = ""

  ## If true, no CREATE DATABASE queries will be sent.  Set to true when using
  ## Telegraf with a user without permissions to create databases or when the
  ## database already exists.
   skip_database_creation = true

  ## Name of existing retention policy to write to.  Empty string writes to
  ## the default retention policy.  Only takes effect when using HTTP.
  # retention_policy = ""

  ## Write consistency (clusters only), can be: "any", "one", "quorum", "all".
  ## Only takes effect when using HTTP.
  # write_consistency = "any"

  ## Timeout for HTTP messages.
   timeout = "30s"
```

Abbildung 5-1: Auszug aus dem InfluxDB Output Plug-in der Konfigurationsdatei „Telegraf.conf“

Des Weiteren wird die Zieldatenbank, in welche die Daten aus den CSV-Dateien geschrieben werden sollen, angegeben. Da in der *InfluxDB* mithilfe des InfluxQL-Befehls „`CREATE DATABASE`“ vorab bereits eine Datenbank mit dem Namen „thesisKS“ für diesen Anwendungsfall erstellt wurde, wird diese in der Konfigurationsdatei als Zieldatenbank festgelegt (database = „thesisKS“). Aufgrund der bereits existierenden Datenbank muss der *Telegraf* Agent keinen „`CREATE DATABASE`“ Befehl senden, sodass die Default-Einstellung bei „skip_database_creation“ auf „true“ bleibt. Dahingegend wird das Timeout von HTTP Nachrichten von 15 Sekunden in den Default-Einstellungen auf 30 Sekunden erhöht. In der Abbildung 5-1 ist ein Auszug aus dem *InfluxDB* Output

Plug-in aus der Konfigurationsdatei mit den zuvor beschriebenen Einstellungen zu sehen. Alle weiteren Konfigurationsmöglichkeiten des *InfluxDB* Output Plug-ins bleiben unverändert auf ihren Default-Werten.

Für die Konfiguration des Inputs wird im „File“ Plug-in ([[inputs.file]]), welches in der Abbildung 5-2 zu sehen ist zunächst der Pfad, unter welchem die in die *InfluxDB* zu schreibenden Dateien zu finden sind (files = [“thesis/inputCSV/**.csv“]) sowie der Datentyp der Dateien, in diesem Fall CSV (data_format = „csv“), angegeben. Zumal der *Telegraf* Agent fortlaufend alle Dateien unter dem angegeben Pfad liest und in die *InfluxDB* schreibt, werden die 27 CSV-Dateien mit den Apple Health Daten nacheinander für den Schreibprozess unter dem genannten Pfad abgelegt. Anschließend sind in Abhängigkeit des Datentyps unterschiedliche Konfigurationen vorzunehmen. Beim CSV-Format ist zum einen anzugeben, wie viele Zeilen Header in den einzulesenden Dateien zu finden sind, was in den vorverarbeiteten CSV-Dateien einer Zeile entspricht (csv_header_row_count = 1). Da die Header in den CSV-Dateien in der ersten Zeile und ersten Spalte beginnen, müssen keine Zeilen oder Spalten für das Lesen der Header Informationen in den CSV-Dateien ausgelassen werden. Im „File“ Input Plug-in in der *Telegraf*-Konfigurationsdatei werden daher folgende Einstellungen gemacht: csv_skip_rows = 0 und csv_skip_columns = 0. Dadurch, dass die CSV-Dateien kommagetrennt sind, wird in der Konfigurationsdatei “,“ als Trennzeichen definiert.

```
# # Reload and gather from file[s] on telegraf's interval.
[[inputs.file]]
#   ## Files to parse each interval.
#   ## These accept standard unix glob matching rules, but with the addition of
#   ## ** as a "super asterisk". ie:
#   ##   /var/log/**.log     -> recursively find all .log files in /var/log
#   ##   /var/log/*/*.log    -> find all .log files with a parent dir in /var/log
#   ##   /var/log/apache.log -> only read the apache log file
   files = ["thesis/inputCSV/**.csv"]
#
#   ## The dataformat to be read from files
#   ## Each data format has its own unique set of configuration options, read
#   ## more about them here:

#   ## https://github.com/influxdata/telegraf/blob/master/docs/DATA_FORMATS_INPUT.md
   data_format = "csv"
   csv_header_row_count = 1
   csv_skip_rows = 0
   csv_skip_columns = 0
   csv_delimiter = ","
   csv_tag_columns = ["userID", "deviceAnonymous", "userDeviceSourceID"]
   csv_measurement_column = "measurement"
   csv_timestamp_column = "time"
   csv_timestamp_format = "Mon Jan 2 15:04:05 -0700 MST 2006"
```

Abbildung 5-2: „File“ Input Plug-in der Konfigurationsdatei „Telegraf.conf“

Des Weiteren können Spalten der CSV-Dateien definiert werden, die in der *InfluxDB* als Tags gespeichert werden (csv_tag_columns). Tags sind optional und enthalten Metadaten. Alle weiteren Spalten der CSV-Dateien, abgesehen vom Zeitstempel („timestamp“) und den Messwerten („measurement“) werden automatisch als Fields behandelt, die aus Metadaten und den zugehörigen Datenwerten bestehen. Der Unterschied zwischen Tags und Fields liegt dabei darin, dass Tag Key-Value Paare indiziert

sind und sich daher insbesondere für die Speicherung von Metadaten eignen, die häufig abgefragt werden, während Field Key-Value Paare nicht indiziert sind. Abfragen auf Tags sind daher deutlich performanter als Abfragen auf Fields, da bei Abfragen auf Fields immer alle Datenpunkte des spezifizierten Zeitintervalls abgefragt werden.[180] Aus diesem Grund werden die Spalten "userID", "deviceAnonymous" und "userDeviceSourceID" als Tags verwendet, zumal Abfragen auf bestimmte User, Gerätetypen oder auch einzelne Geräte im vorliegenden Anwendungsfall üblich sind und eine gute Performance aufweisen sollten. Weiterhin sind die Spalten der CSV-Dateien anzugeben, aus denen der Name der Messwerte (csv_measurement_column = "measurement") und der Zeitstempel der Messwerte (csv_timestamp_column = "time") zu entnehmen sind. Für den Zeitstempel ist zudem auch das Zeitformat, in dem der Zeitstempel in den CSV-Dateien gespeichert ist, zu definieren (csv_timestamp_format = "Mon Jan 2 15:04:05 -0700 MST 2006").

Ein wesentlicher Bestandteil der Konfiguration von *Telegraf* ist zudem die Konfiguration des Agenten selbst, die in der „Telegraf.conf“-Datei unter [agent] erfolgt. Sie umfasst u.a. die Angabe des Intervalls, in dem die Daten aus den Input-Quellen gesammelt werden („interval“), und die Größe der Batches, in denen Metriken an die verschiedenen Outputs gesendet werden („metric_batch_size“). Diese Einstellungen werden unter Berücksichtigung der in die *InfluxDB* zu schreibenden Datenmenge und der im Anwendungsfall zur Verfügung stehenden Hardware (vgl. Kapitel 4.1.1) konfiguriert. Da insgesamt über drei Millionen Aktivitätsmessungen in die *InfluxDB* zu schreiben sind, soll die Batch Größe möglichst hoch gewählt werden. Allerdings sollen die in Intervallen durchgeführten Schreibvorgänge aufgrund zu hoher Batch Größen nicht fehlschlagen und den ebenfalls konfigurierbaren Pufferspeicher („metric_buffer_limit“) nicht zum Überlaufen bringen. Vor diesem Hintergrund ist bei der Konfiguration ebenfalls der in der Hardware Sizing Guideline angegebene Wert zu den Field Writes pro Sekunde zu beachten, der für die angenommene niedrige Belastung (vgl. Kapitel 4.2.1) weniger als 5.000 Field Writes pro Sekunde entspricht. Zumal für den praktischen Vergleich ein HDD- und nicht, wie in der Hardware Sizing Guideline empfohlen, ein Solid State Drive (SSD)-Speicher vorliegt und daher eine geringere Performance zu erwarten ist, sollte mit einer niedrigeren Anzahl als 5.000 Field Writes pro Sekunde gerechnet werden.[181] In der Abbildung 5-3 sind die für den praktischen Vergleich verwendeten Konfigurationseinstellungen des *Telegraf* Agenten zu sehen, die anlässlich und unter Berücksichtigung der zuvor beschriebenen Gegebenheiten sowie aufgrund der Erfahrungen aus testweise durchgeführten Schreibvorgängen ausgewählt wurden. In den Testläufen wurden verschiedene Einstellungen der unterschiedlichen Konfigurationsparameter untersucht. Dabei konnte beobachtet werden, dass Schreibvorgänge bei der Wahl einer zu hohen „metric_batch_size“ oder einem zu niedrigen „interval“ und einem zu

[180] Vgl. Bang (2019c).

[181] Vgl. InfluxData (2019c).

niedrigen „metric_buffer_limit“ vermehrt fehlschlagen („HTTP 413 Request Entity Too Large“) und bzw. oder es zu einem überlaufenden Pufferspeicher kommt („Metric buffer overflow“).

```
# Configuration for telegraf agent
[agent]
  ## Default data collection interval for all inputs
  interval = "60s"
  ## Rounds collection interval to 'interval'
  ## ie, if interval="10s" then always collect on :00, :10, :20, etc.
  round_interval = true

  ## Telegraf will send metrics to outputs in batches of at most
  ## metric_batch_size metrics.
  ## This controls the size of writes that Telegraf sends to output plugins.
  metric_batch_size = 50000

  ## For failed writes, telegraf will cache metric_buffer_limit metrics for each
  ## output, and will flush this buffer on a successful write. Oldest metrics
  ## are dropped first when this buffer fills.
  ## This buffer only fills when writes fail to output plugin(s).
  metric_buffer_limit = 500000

  ## Collection jitter is used to jitter the collection by a random amount.
  ## Each plugin will sleep for a random time within jitter before collecting.
  ## This can be used to avoid many plugins querying things like sysfs at the
  ## same time, which can have a measurable effect on the system.
  collection_jitter = "20s"

  ## Default flushing interval for all outputs. Maximum flush_interval will be
  ## flush_interval + flush_jitter
  flush_interval = "60s"
  ## Jitter the flush interval by a random amount. This is primarily to avoid
  ## large write spikes for users running a large number of telegraf instances.
  ## ie, a jitter of 5s and interval 10s means flushes will happen every 10-15s
  flush_jitter = "20s"
```

Abbildung 5-3: Auszug aus der Konfiguration des Telegraf Agenten in der Datei „Telegraf.conf“

Neben den in der Abbildung 5-3 dargestellten Einstellungen, kann in der „Telegraf.conf“-Datei ebenfalls definiert werden, ob Debug-Meldungen gesendet werden, oder nicht. Um das Debugging im Rahmen der Implementierung der *InfluxDB* zu erleichtern, wird das Senden von Debug-Meldungen aktiviert („debug = true“). Für die übrigen Konfigurationseinstellungen werden die Default-Werte verwendet.

Als weitere Komponente der *InfluxData* Plattform wird neben dem *Telegraf* Agenten ebenfalls *Chronograf* verwendet. *Chronograf* ist ein administratives UI, mit dem die in der *InfluxDB* gespeicherten Daten eingesehen, abgefragt und in Echtzeit visualisiert werden können. Für die Visualisierung der Daten stehen verschiedene Templates und Sammlungen von vorgefertigten Dashboards zur Verfügung, die einfach und schnell adaptiert werden können.[182] Der Download und die Installation erfolgte auch bei *Chronograf* gemäß der Beschreibung in der Dokumentation.[183] Eine benutzerdefinierte Kon-

[182] Vgl. InfluxData (2019a).

[183] Vgl. Anderson (2019).

figuration von *Chronograf* kann, wie auch bei *InfluxDB* und *Telegraf*, über eine Konfigurationsdatei vorgenommen werden. Da der Default-Port von *Chronograf* (Port Nummer 8888) zum Zeitpunkt der Konfiguration auf dem Server schon vergeben ist, wird in der Konfigurationsdatei mit einem Key-Value Paar der Port mit der Nummer 8800 als Port für *Chronograf* definiert, sodass das *Chronograf* UI über http://localhost:8800 bzw. http://139.6.56.162:8800 über den Webbrowser aufgerufen werden kann. Über das Web-UI kann dann eine Verbindung zur *InfluxDB* aufgebaut werden, indem die Adresse der entsprechenden *InfluxDB* (http://localhost:8082) sowie die zu verbindende Datenbank, in diesem Fall „thesisKS“, angegeben wird, wie in der Abbildung 5-4 zu sehen ist. Im nächsten Schritt können bereits Standard-Dashboards für die erstellte Verbindung ausgewählt werden. Im konkreten Fall werden automatisch zwei Dashboards, eins für die *InfluxDB* und eins für das System, vorgeschlagen.

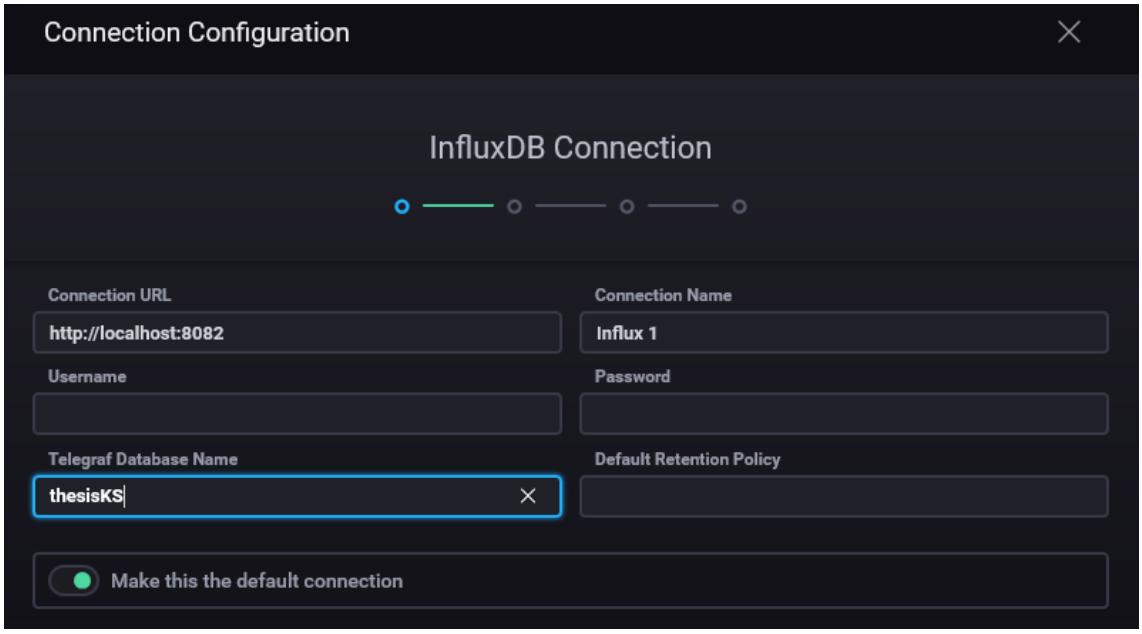

Abbildung 5-4: Konfiguration der Verbindung von Chronograf zur InfluxDB

Damit in den Dashboards Daten angezeigt werden und um *Chronograf* schließlich als UI für die Abfrage und Visualisierung von den in der *InfluxDB* gespeicherten Daten nutzen zu können, ist es neben der eigentlichen Konfiguration von *Chronograf* notwendig, den *Telegraf* Agenten für die Sammlung der in der *InfluxDB* gespeicherten Daten zu konfigurieren. Hierzu ist in der Konfigurationsdatei von *Telegraf* beim Input Plug-in der *InfluxDB* ([[inputs.InfluxDB]]) lediglich die URL anzugeben, von wo *Telegraf* die Daten sammeln soll (urls = [http://localhost:8082/debug/vars“]). Darüber hinaus wird, wie bei der Konfiguration unter [[ouputs.InfluxDB]], das Timeout von HTTP Nachrichten von 15 Sekunden in den Default-Einstellungen auf 30 Sekunden erhöht. Zumal in den empfohlenen Standard-Dashboards neben Daten aus der *InfluxDB* zusätzliche Daten, bspw. über die CPU-Auslastung oder die Systemlast, abgefragt und visualisiert werden, werden auch folgende für die Sammlung der relevanten Daten notwendige Input

Plug-ins in der Konfigurationsdatei von *Telegraf* konfiguriert: system, net, disk, mem, cpu, processes, diskio und swap.

Um die Performance der *InfluxDB* in Kombination mit dem *Telegraf* Agenten beim Schreiben von Daten in die *InfluxDB* anhand der Kennzahl „Anzahl der geschriebenen Metriken pro Sekunde“ beurteilen und im nächsten Schritt mit dem Event Store *Event Store* vergleichen zu können, werden neben dem zuvor genannten Schreibvorgang in die Datenbank „thesisKS“ weitere Schreibvorgänge mit unterschiedlichen Batch Größen durchgeführt (vgl. Kapitel 4.3.2). Die Batch Größen reichen von 10.000 Metriken bis 80.000 Metriken je Batch, aufsteigend in Schritten von 10.000 Metriken. Die Daten werden dabei jeweils in neu erstellte Datenbanken geschrieben. Alle weiteren Konfigurationseinstellungen bleiben hingegen unverändert. Je Batch Größe wurden mindestens 200 Batches in die *InfluxDB* geschrieben, um eine durchschnittliche Anzahl der geschriebenen Metriken pro Sekunde ermitteln zu können, die nicht von extern beeinflussten Vorfällen bei einzelnen Schreiboperationen verfälscht ist. Das Ergebnis der im Rahmen der Performance-Analyse durchgeführten Schreibvorgänge mit unterschiedlichen Batch Größen ist übersichtlich in der Tabelle 5-1 zu sehen.

Batch Größe	**10.000**	**20.000**	**30.000**	**40.000**	**50.000**	**60.000**	**70.000**	**80.000**	**Gesamt**
Anzahl Batch Writes	251	250	225	216	216	218	229	212	**1.817**
ø Anzahl geschriebener Metriken pro Sekunde (Mittelwert)	25.531	30.843	24.831	28.040	30.228	28.987	28.121	22.549	**27.425**

Tabelle 5-1: Durchschnittliche Performance der Schreibvorgänge unterschiedlicher Batch Größen in die InfluxDB

Die durchschnittlich meisten Metriken, nämlich 30.843 Metriken je Sekunde, konnten unter den gegebenen Voraussetzungen bei einer Batch Größe von 20.000 Metriken in die *InfluxDB* geschrieben werden. Im Gegensatz dazu konnten mit einer Batch Größe von 80.000 nur 22.549 Metriken je Sekunde und somit durchschnittlich die wenigsten Metriken in die *InfluxDB* geschrieben werden. Der Durchschnitt über alle Batch Größen hinweg beträgt bei den im Rahmen der Performance-Analyse durchgeführten Schreibvorgängen 27.425 Metriken je Sekunde. Werden die ermittelten Durchschnittswerte der unterschiedlichen Batch Größen betrachtet, kann keine eindeutige Abhängigkeit zwischen der Batch Größe und der durchschnittlichen Anzahl geschriebener Metriken je Sekunde, wie bspw. eine steigende durchschnittliche Anzahl geschriebener Metriken pro Sekunde bei einer steigenden Batch Größe, erkannt werden.

Der Boxplot in der Abbildung 5-5 veranschaulicht die Verteilung der Anzahl der geschriebenen Metriken der einzelnen Schreiboperationen je Batch Größe. Hieraus kann abgelesen werden, dass bei einer Batch Größe von 20.000 Metriken je Batch im Rahmen des praktischen Vergleichs die meisten Metriken pro Sekunde in die *InfluxDB* geschrieben werden konnten (Maximum), während bei einer Batch Größe von 50.000

Metriken die wenigstens Metriken pro Sekunde in die *InfluxDB* geschrieben wurden (Minimum). Auffällig bei der Betrachtung der Abbildung 5-5 ist überdies, dass die Spanne zwischen der minimalen und maximalen Anzahl an geschriebenen Metriken je Sekunde bei allen Batch Größen, insbesondere jedoch bei 50.000 Metriken je Batch, sehr groß ist. Aus diesem Grund sollten die unter den gegebenen Voraussetzungen ermittelten Durchschnittswerte der Anzahl der in die *InfluxDB* geschriebenen Metriken pro Sekunde (vgl. Tabelle 5-1) eher als Tendenz betrachtet werden.

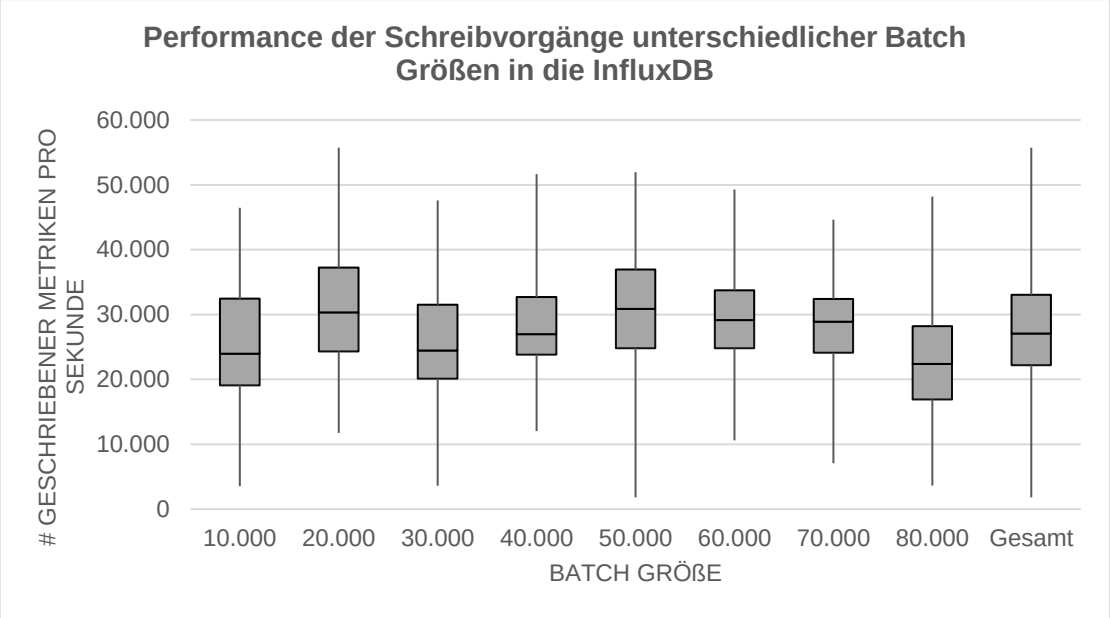

Abbildung 5-5: Performance der Schreibvorgänge unterschiedlicher Batch Größen in die InfluxDB

5.1.2 Event Store

Als Event Store wird für die Durchführung des praktischen Vergleichs der *Event Store* Version 5.0.1 verwendet. Der Download und die Installation erfolgten gemäß der Anleitung für Linux Ubuntu in der Dokumentation. Eine Verbindung zum *Event Store* kann u.a. mit dem Admin UI oder mittels der HTTP API über die Port Nummer 2113 aufgebaut werden. Damit der *Event Store* nicht nur vom Server selbst, sondern auch von extern erreichbar ist, wird in der Konfigurationsdatei „eventstore.conf“ die IP-Adresse des Servers (139.6.56.162, vgl. Kapitel 4.1.1) als interne sowie als externe IP-Adresse hinterlegt, wie in der Abbildung 5-6 zu sehen ist. Nach erfolgreicher Installation und Konfiguration kann der Event Store mit dem Befehl „`sudo systemctl start eventstore`“ gestartet werden.[184]

[184] Vgl. Chinchilla (2019a).

```
---
IntIp: 139.6.56.162
ExtIp: 139.6.56.162
---
```

Abbildung 5-6: Inhalt der Konfigurationsdatei „eventstore.conf“

Das Schreiben von Events in Event Streams des *Event Stores* ist sowohl über eine manuelle Eingabe im Admin UI als auch über die HTTP API möglich. Wird ein Event in einen Event Stream, der bisher noch nicht existiert hat, geschrieben, wird der Event Stream implizit erstellt. Im Rahmen des praktischen Vergleichs wird die HTTP API für das Schreiben der Events verwendet. Über die HTTP API können Events in unterschiedlichen Dateiformaten, nämlich als JSON- oder XML-Datei, in den *Event Store* geschrieben werden. Ein Event bzw. die Daten, die ein Event repräsentieren, bestehen dabei immer aus einer „eventId“ in Form eines Universally Unique Identifiers (UUID), dem „eventType“ sowie den zum eigentlichen Event zugehörenden Daten ("data") und Metadaten („metadata“). Je nach Schema in den JSON- bzw. XML-Dateien kann entweder ein einzelnes Event oder mehrere Events auf einmal in sogenannten Batch Writes mit einem einzigen POST Request in den *Event Store* geschrieben werden. Die folgende Abbildung 5-7 zeigt den für einen Batch Write notwendigen Aufbau einer JSON-Datei des Typs „application/vnd.eventstore.events+json“, bei dem mehrere Events, in diesem Fall zwei, innerhalb eines Arrays platziert sind.[185]

```
[
    {
        "eventId": "fbf4b1a1-b4a3-4dfe-a01f-ec52c34e16e4",
        "eventType": "event-type",
        "data": {
            "a": "1"
        }
    },
    {
        "eventId": "0f9fad5b-d9cb-469f-a165-70867728951e",
        "eventType": "event-type",
        "data": {
            "b": "2"
        }
    }
]
```

Abbildung 5-7: Schema einer JSON-Datei des Typs „application/vnd.eventstore.events+json“ für Batch Writes in den Event Store (Quelle: (Chinchilla, 2018a))

Zumal in den für den praktischen Vergleich zur Verfügung stehenden Daten mehr als drei Millionen Aktivitätsmessungen und somit Events vorliegen und nicht jedes einzelne dieser Events mit einem eigenen POST Request in den *Event Store* geschrieben wer-

[185] Vgl. Chinchilla (2018a).

den soll, werden Batch Writes mit mehreren Events verwendet. Bei solchen Batch Writes von vielfachen Events in einem einzigen POST Request ist zu beachten, dass die Events entweder vollständig in den angegebenen Event Stream des *Event Stores* geschrieben werden oder der Schreibvorgang als Ganzes fehlschlägt.[186] Vor diesem Hintergrund werden die mehr als drei Millionen Aktivitätsmessungen bzw. Events nicht in einem einzigen Batch Write in den *Event Store* geschrieben, sondern in mehrere kleinere Batches je User aufgeteilt. Die Größe der Batches, d.h. die maximale Anzahl an Events, die in einem POST Request in den *Event Store* geschrieben wird, kann als Variabel im *R*-Skript „EventStore-loop-dataPreparation.R“ (s. Anhang D) definiert werden.

Der Vorgang, die Apple Health Daten aus den XML-Daten in das in der Abbildung 5-7 gezeigte für Batch Writes geforderte Format zu bringen, kann in zwei Schritte unterteilt werden und ist in der Funktion „EventStoredataPreparation“ in der Datei „EventStoredataPreparation.R“ (s. Anhang E) programmiert. Diese Funktion wird mit dem zuvor genannten *R*-Skript „EventStore-loop-dataPreparation.R“ (s. Anhang D) aufgerufen und auf alle „Export.xml“-Dateien im Ordner „export“ auf dem Server angewendet.

Im ersten Schritt der Funktion „EventStoredataPreparation“ (s. Bereich unter „# basic data preparation“ in der zuvor genannten Datei) werden die Daten mit der unter Kapitel 5.1 beschriebenen Funktion zur allgemeinen Datenaufbereitung („dataPreparation“) aufbereitet. Diese Funktion wird geringfügig angepasst, indem die Datumsformate des „startDate“s und „endDate“s nicht als Dezimalzahl des POSIXct-Formates, sondern als Text („character“) formatiert werden. Als Ergebnis dieses ersten Schrittes der Funktion „EventStoredataPreparation“ liegen die für die Speicherung in das DBMS vorgesehenen Daten formatiert in einem Dataframe vor.

Im zweiten Schritt (s. Bereich unter „# preparation for JSON“ in der Datei „EventStoredataPreparation.R“ im Anhang E) erfolgt daraufhin die eigentliche Umwandlung der Apple Health Daten aus dem Dataframe in das in der Abbildung 5-7 gezeigte Format für Batch Writes. Bevor diese Umwandlung jedoch erfolgen kann, ist zunächst zu definieren, wie eine Datenzeile des Dataframes schematisch im *Event Store* gespeichert werden sollen. Zumal je User eine XML-Datei mit den jeweiligen Aktivitätsdaten des Users vorliegt, soll für jeden User ein eigener Event Stream, d.h. insgesamt 27 Event Streams, im *Event Store* angelegt werden. Während die UUID als erster Bestandteil eines Events in den Apple Health Daten nicht hinterlegt ist und daher im Rahmen der Umwandlung der Daten in das geforderte Format neu zu generieren ist, soll der „eventType“ von den in den Apple Health Daten enthaltenen Aktivitätsmessungen („measurements“), z.B. „DistanceWalkingRunning“ und „StepCount“ (vgl. Kapitel 4.1.2), dargestellt werden. Als Daten („data“) soll der Wert der Aktivitätsmessung („value“), der Start- und Endzeitpunkt der Messung („startDate“ und „endDate“) sowie die im Data-

[186] Vgl. Chinchilla (2018a).

frame enthalten Informationen zum Gerät des Users („deviceAnonymous“ und „userDeviceSourceID“) in den *Event Store* geschrieben werden. Als Metadaten („metadata“) soll lediglich die Einheit der Messung („unit“) gespeichert werden.

Unter Berücksichtigung dieses beschriebenen Schemas werden als Grundlage für die weitere Datenverarbeitung mit der Programmiersprache *R* Klassen für die Events sowie deren Bestandteile „data“ und „metadata“ definiert (s. *R*-Skript „EventStore-loop-dataPreparation.R“ im Anhang D). Unter Verwendung dieser Klassen wird folglich eine Funktion entwickelt, welche die Daten einer Zeile des Dataframes in das für ein Event vorgesehene Format, bestehend aus der „eventId“, des „eventType“s sowie den Daten („data“) und Metadaten („metadata“) umwandelt. Diese Funktion mit der Bezeichnung „SingleEventPreparation“ ist in der folgenden Abbildung 5-8 oder im *R*-Skript „EventStore-SingleEventPreparation.R“ im Anhang F zu sehen.

```
SingleEventPreparation <- function(inputDataframe, rowIndex) {
  event.metadata <- new("event.metadata",
                        unit = inputDataframe[rowIndex,"unit"])
  event.data <- new("event.data",
                    value = inputDataframe[rowIndex,"value"],
                    startDate = inputDataframe[rowIndex,"startDate"],
                    endDate = inputDataframe[rowIndex,"endDate"],
                    deviceAnonymous = inputDataframe[rowIndex,"deviceAnonymous"],
                    userDeviceSourceID = inputDataframe[rowIndex,"userDeviceSourceID"])
  event <- new("event", eventId= UUIDgenerate(),
               eventType = inputDataframe[rowIndex,"measurement"],
               data = event.data ,
               metadata = event.metadata)
  event
}
```

Abbildung 5-8: Funktion „SingleEventPreparation“ in R

Nachdem im zweiten Schritt der Funktion „EventStoredataPreparation“ als Teil der „# preparation for JSON“ zunächst die Anzahl der notwenigen Batches je XML-Datei auf Basis der festgelegten Batch Größe berechnet wird, erfolgt für jedes Batch schließlich die Datenumwandlung der Apple Health Daten des Dataframes. Die einzelnen Datenzeilen eines Batchs werden mit der definierten Funktion „SingleEventPreparation“ zuerst in das geforderte Event-Format transformiert, daraufhin in das JSON-Format konvertiert, in einen Vektor geschrieben und als letzter Schritt der Funktion „EventStoredataPreparation“ gesamthaft je Batch als JSON-Datei mit dem in der Abbildung 5-7 gezeigten Schema gespeichert.

Für den quantitativen Vergleich sollen mehrere Schreibvorgänge mit unterschiedlichen Batch Größen durchgeführt werden (vgl. Kapitel 4.3.2). Damit kann zum einen die durchschnittliche Anzahl der geschriebenen Events pro Sekunde je Batch Größe ermittelt und der Einfluss der Batch Größe auf diese Kennzahl untersucht werden. Zum anderen ist es dadurch möglich, die maximale Batch Größe zu ermitteln, bei der die Schreibvorgänge im gegebenen Anwendungsfall mit der zur Verfügung stehenden Hardware zuverlässig, ohne Fehlermeldung erfolgen.

Wie bei der *InfluxDB* sollen auch beim *Event Store* bei den im Rahmen des praktischen Vergleichs mehrfach durchzuführenden Schreibvorgängen insgesamt mindestens 200 Batches je Batch Größe in den *Event Store* geschrieben werden. Um eine Vergleichbarkeit herzustellen, reichen die Batch Größen auch hier von 10.000 Events bis 80.000 Events je Batch, aufsteigend in Schritten von 10.000. Da die Datenmenge bzw. die Events je User variieren (vgl. Kapitel 4.1.2), variiert auch die notwendige Anzahl an Batches, um die gesamten Daten eines Users in den *Event Store* zu schreiben. Während bei einer Batch Größe von 10.000 Events für das Schreiben der Daten aller User insgesamt 341 Batches notwendig sind, reichen bei einer Batch Größe von 80.000 Events schon 55 Batches aus, wie übersichtlich in der Tabelle 5-2 oder detailliert im Anhang G zu sehen ist. Da die Anzahl der notwendigen Batches bei diesen großen Batch Größen etwa 60 beträgt, sollen auch für die Batch Größen kleiner als 60.000 Events nur etwa 60 Batches je Batch Größe je Schreibvorgang in den *Event Store* geschrieben werden. Zumal für die Performance-Analyse jedoch, wie zuvor beschrieben, insgesamt mindestens 200 Batches je Batch Größe in den Event Store geschrieben werden sollen, variiert die Anzahl der durchzuführenden Schreibvorgänge je Batch Größe.

Batch Größe in Events	**10.000**	**20.000**	**30.000**	**40.000**	**50.000**	**60.000**	**70.000**	**80.000**
Anzahl notwendiger Batches	341	177	125	96	80	68	62	55

Tabelle 5-2: Anzahl der notwendigen Batches je Batch Größe

Da die Batch Größe in der zuvor beschriebenen Funktion „EventStoredataPreparation“ variabel definiert werden kann, können die zuvor erläuterten Schritte der Datenaufbereitung nacheinander für die unterschiedlichen Batch Größen durchgeführt werden. Als Ergebnis der Datenaufbereitung liegen je Batch Größe schließlich etwa 60 Batches mit aufbereiteten Apple Health Daten vor, die als JSON-Dateien mit dem Schema „application/vnd.eventstore.events+json“ auf dem Server gespeichert sind.

```
curl -i -d "@multiple-events.json" "http://127.0.0.1:2113/streams/newstream"
-H "Content-Type:application/vnd.eventstore.events+json"
```

Abbildung 5-9: POST Request für Batch Writes in den Event Store

Die Abbildung 5-9 zeigt einen POST Request, mit dem mehrere Batches von Events des Typs „application/vnd.eventstore.events+json“ in den *Event Store* geschrieben werden können. Die Bezeichnung der JSON-Datei, in der die in den *Event Store* zu schreibenden Events enthalten sind, ist im POST Request in der Abbildung 5-9 „multiple-events.json“, „127.0.0.1“ ist die IP-Adresse, unter welcher der *Event Store* zu erreichen ist, und „newstream“ ist der Name des Streams, in welchen die Events geschrieben werden sollen. Damit im Rahmen des praktischen Vergleichs nicht für jedes der vorbereiteten Batches manuell ein eigener POST Request geschrieben werden und

einzeln gesendet werden muss, wurde ein weiteres *R*-Skript („EventStore-POST-batchWrite“, s. Anhang H) erstellt. Mit diesem *R*-Skript wird je Schreibvorgang und Batch Größe für jede der in den *Event Store* zu schreibenden JSON-Dateien ein POST Request generiert. All diese POST Requests eines Schreibvorgangs werden dann in eine Datei (standardmäßig „batchWrite.sh“) geschrieben. Um alle Batches eines Schreibvorgangs in den *Event Store* zu schreiben, muss somit lediglich diese generierte Datei, bspw. „batchWrite.sh“, mit dem Befehl „`sudo bash batchWrite.sh`“ im Ubuntu Terminal ausgeführt werden. Für jeden erfolgreich ausgeführten POST Request wird eine Antwort mit dem HTTP Status 201 Created erhalten, die ebenfalls den Speicherort („Location“) im *Event Store* beinhaltet. Als Speicherort ist in der HTTP Antwort über den Event Stream hinaus auch die Nummer des Events innerhalb des Event Streams angegeben, unter welchem das erste Event des jeweiligen Batchs gespeichert wurde. Dabei ist zu beachten, dass die Event Streams nicht bei Event Nummer Eins, sondern bei Event Nummer Null beginnen. In der Abbildung 5-10 ist beispielhaft die Antwort auf den POST Request „`curl -i -d "@User1-batch-1.json" http://139.6.56.162:2113/streams/User1thesisKS" -H "Content-Type: application/vnd. eventstore.events+json"`“ zu sehen.

```
HTTP/1.1 201 Created
Access-Control-Allow-Methods: POST, DELETE, GET, OPTIONS
Access-Control-Allow-Headers: Content-Type, X-Requested-With, X-Forwarded-Host, X-
Forwarded-Prefix, X-PINGOTHER, Authorization, ES-LongPoll, ES-ExpectedVersion, ES-
EventId, ES-EventType, ES-RequiresMaster, ES-HardDelete, ES-ResolveLinkTos
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: Location, ES-Position, ES-CurrentVersion
Location: http://139.6.56.162:2113/streams/User1thesisKS/0
Content-Type: text/plain; charset=utf-8
Server: Mono-HTTPAPI/1.0
Date: Sun, 23 Jun 2019 09:52:34 GMT
Content-Length: 0
Keep-Alive: timeout=15,max=100
```

Abbildung 5-10: Antwort HTTP 201 Created auf erfolgreichen POST Request

Für die Performance-Analyse im Rahmen des praktischen Vergleichs wurden für die Batch Größen von 10.000 bis 80.000 Events schließlich jeweils insgesamt mindestens 200 Batches je Batch Größe mithilfe von mehreren Schreibvorgängen mit jeweils etwa 60 Batches in den *Event Store* geschrieben. Für jeden Schreibvorgang wurde die durchschnittliche Anzahl der geschriebenen Events pro Sekunde ermittelt und anschließend ein Mittelwert je Batch Größe sowie ein Mittelwert über alle Batch Größen gebildet. Das Ergebnis ist in der Tabelle 5-3 zu sehen.[187]

[187] Detaillierte Informationen und die Datengrundlage für die Performance-Analyse sind in der Datei „Performance-Analyse Event Store.xlsx“ auf der beigelegten CD zu finden.

Batch Größe	Nr. Schreibvorgang	Anzahl erfolgreicher & [Anzahl gesamter] Batch Writes mit jeweiliger Batch Größe	ø Anzahl geschriebener Events pro Sekunde (Mittelwert)	ø Anzahl geschriebener Events pro Sekunde je Batch Größe (gewichteter Mittelwert)	ø Anzahl geschriebener Events pro Sekunde (gewichteter Mittelwert)
10.000	1	58 [58]	8.534	**7.522**	
	2	58 [58]	8.190		
	3	58 [58]	7.500		
	4	58 [58]	5.862		
20.000	1	55 [55]	8.030	**7.864**	
	2	55 [55]	8.364		
	3	55 [55]	7.727		
	4	55 [55]	7.333		
30.000	1	55 [55]	8.682	**8.330**	
	2	55 [55]	8.800		
	3	55 [55]	8.455		
	4	55 [55]	7.382		
40.000	1	55 [55]	8.339	**7.984**	
	2	55 [55]	8.068		
	3	55 [55]	8.255		
	4	55 [55]	7.273		
50.000	1	48 [48]	8.373	**8.012**	
	2	48 [48]	8.310		
	3	48 [48]	8.142		
	4	48 [48]	7.221		
	5	48 [48]	7.825		
60.000	1	41 [41]	8.415	**8.536**	
	2	41 [41]	8.098		
	3	40 [41]	8.342		
	4	41 [41]	9.284		
	5	41 [41]	8.664		
70.000	1	33 [35]	8.263	**7.910**	
	2	31 [35]	8.204		
	3	35 [35]	8.516		
	4	32 [35]	8.555		
	5	26 [35]	5.857		
	6	30 [35]	7.778		
	7	32 [35]	7.747		
80.000	1	22 [28]	8.348	**7.926**	
	2	12 [28]	7.293		
	3	26 [28]	8.600		
	4	21 [28]	7.827		
	5	8 [28]	6.727		
	6	15 [28]	7.935		
	7	9 [28]	8.135		
	8	14 [28]	7.697		
	9	22 [28]	8.004		
	10	19 [28]	7.749		
	11	21 [28]	7.925		
	12	15 [28]	7.620		

Tabelle 5-3: Durchschnittliche Performance der Schreibvorgänge unterschiedlicher Batch Größen in den Event Store

Anders als bei der *InfluxDB* wird bei der Auswertung der Performance beim Schreiben in den *Event Store* lediglich der Mittelwert der Kennzahl „Anzahl der geschriebenen Events pro Sekunde“ in der Tabelle 5-3 ausgewiesen. Dies ist darauf zurückzuführen, dass die Dauer für das Schreiben eines Batchs in den *Event Store* lediglich in der Einheit Sekunde vorliegt und es dadurch bei der Betrachtung einzelner Schreiboperationen, wie es bspw. für die Berechnung für den Median oder das Minimum und Maximum nötig ist, zu Ungenauigkeiten infolge der zu groben Zeiteinheit kommt.

Für die Ermittlung der durchschnittlichen Anzahl der geschriebenen Events pro Sekunde je unterschiedlicher Batch Größe wurden nur diejenigen Batch Writes berücksichtigt, bei denen die Anzahl der im Batch enthaltenen und erfolgreich in den *Event Store* geschriebenen Events der jeweiligen Batch Größe entspricht. Auch bei der Berechnung der Durchschnitte je Batch Größe und über alle Batch Größen wurde die Anzahl der durchschnittlich geschriebenen Events pro Sekunde mit der Anzahl der erfolgreich geschriebenen Batches der jeweiligen Batch Größe gewichtet. Die genauen Anzahlen der erfolgreich durchgeführten Batch Writes je Batch Größe und Schreibvorgang sind aus der Tabelle 5-3 zu entnehmen. Dass diese Anzahl der erfolgreich in den *Event Store* geschriebenen Batches mit der jeweiligen Batch Größe, insbesondere bei den großen Batch Größen, von den je Schreibvorgang etwa 60 in den *Event Store* zu schreibenden Batches abweicht, hat mehrere Gründe. Einerseits liegt es an den unterschiedlichen Datenmengen je User, die dazu führen, dass bei einer Batch Größe von 80.000 lediglich 28 der 55 notwendigen Batches auch wirklich 80.000 Events enthalten (vgl. Werte in eckiger Klammer in der Tabelle 5-3). Andererseits kommt es bei den Schreibvorgängen mit Batch Größen größer gleich 60.000 regelmäßig zum Fehlschlagen einzelner POST Requests mit den Fehlermeldungen „HTTP 408 Server was unable to handle request“ in time oder „HTTP 500 Write Timeout“. Daraus kann abgeleitet werden, dass unter den Gegebenheiten des Anwendungsfalles maximal eine Batch Größe von 50.000 Events je Batch verwendet werden sollte, um die Events zuverlässig erfolgreich in den *Event Store* zu schreiben.[188]

In der Tabelle 5-3 kann abgelesen werden, dass bei einer Batch Größe von 60.000 Events mit durchschnittlich 8.536 die meisten Events je Sekunde in den *Event Store* geschrieben werden konnten. Die durchschnittlich wenigsten Events je Sekunde, nämlich 7.522, wurden mit einer Batch Größe von 10.000 Events in den *Event Store* geschrieben. Der Durchschnitt über alle Batch Größen liegt bei 8.001 geschriebenen Events pro Sekunde. Auffällig ist, dass obwohl bei allen Schreibvorgängen einer Batch Größe die gleichen Batches an Daten in den *Event Store* geschrieben wurden, die durchschnittliche Anzahl an erfolgreich geschriebenen Events pro Sekunde bei allen Batch Größen je Schreibvorgang jedoch stark variiert. Aufgrund dieser Volatilität der

[188] In der Datei „Performance-Analyse Event Store.xlsx“ auf der beigelegten CD ist eine detaillierte Aufstellung der einzelnen POST Requests der durchgeführten Schreibvorgänge je Batch Größe hinterlegt, mit der die Berechnung der Durchschnittswerte nachvollzogen werden kann und aus der ebenfalls hervorgeht, welche POST Requests fehlgeschlagen sind.

unter den gegebenen Voraussetzungen ermittelten Durchschnittswerte sollte die durchschnittliche Anzahl der in den *Event Store* geschriebenen Events pro Sekunde (vgl. Tabelle 5-3) eher als Tendenz betrachtet werden.

Bei der Betrachtung der in der Tabelle 5-3 abgebildeten Durchschnittswerte kann darüber hinaus nicht festgestellt werden, dass die Anzahl der pro Sekunde erfolgreich in den *Event Store* geschriebenen Events bei steigender Batch Größe stetig zu- oder abnimmt. Somit wird keine Abhängigkeit zwischen der Batch Größe und der durchschnittlichen Anzahl geschriebener Events je Sekunde erkannt.

Unabhängig von den zuvor herausgestellten Performance-Aspekten wird eine Batch Größe von 50.000 Events verwendet, um die gesamten Apple Health Daten aller User in den *Event Store* zu schreiben, um im Rahmen des zweiten Teils des praktischen Vergleichs Abfragen auf den Daten durchführen zu können. Die Batch Größe von 50.000 Events ist die größte Batch Größe, bei der alle Batch Writes der durchgeführten Schreibvorgänge je Batch Größe fehlerfrei und erfolgreich erfolgten. Durch die Wahl dieser Batch Größe müssen so wenig JSON-Dateien des Typs „application/vnd.eventstore.events+json“ und folglich auch so wenig POST Requests wie möglich gesendet werden, um die gesamten für den praktischen Vergleich zur Verfügung stehenden Daten in den *Event Store* zu schreiben, nämlich je ein POST Request für jedes der 80 notwendigen Batches (vgl. Tabelle 5-2).

5.1.3 Vergleich der InfluxDB und des Event Stores bei der Implementierung

Die Installation verlief sowohl bei der TSDB *InfluxDB* sowie bei den als Ergänzung zur *InfluxDB* verwendeten und ebenfalls zur *InfluxData* Plattform gehörenden Anwendungen *Telegraf* und *Chronograf* als auch bei dem Event Store *Event Store* problemlos und wie in der veröffentlichen Dokumentation beschrieben.

Bei der Konfiguration unterscheiden sich die beiden DBMS jedoch deutlich. Während die Angabe der IP-Adresse des Servers die einzige manuelle Konfiguration in der Konfigurationsdatei des *Event Stores* ist, ist im Rahmen der Implementierung der *InfluxDB* neben der eigentlichen TSDB *InfluxDB* zusätzlich der *Telegraf* Agent und *Chronograf* zu konfigurieren. Da die Konfiguration des *Telegraf* Agenten Einstellungen zu allen Inputs und Outputs sowie zum Agenten selbst umfasst, ist sie dabei mit dem größten Aufwand verbunden. Hier ist zum einen die Konfiguration des „File“ Input Plug-ins zu nennen, wo festzulegen ist, wie die Apple Health Daten inhaltlich im Schema der *InfluxDB* gespeichert werden sollen, d.h. welche Daten als Tags und welche Daten als Fields gespeichert werden sollen. Zum anderen zeigt sich die Konfiguration des *Telegraf* Agenten, bestehend aus den Einstellungen zum Intervall für die Datensammlung, der Batch Größe, der Größe des Pufferspeichers etc. als besonders aufwendig. Zwar ist in der Hardware Sizing Guideline ein Anhaltspunkt zu der maximalen Anzahl an Field Writes pro Sekunde angegeben, jedoch lassen sich darüber hinaus keine Empfehlungen zu optimalen Einstellungen für das Schreiben von großen Batches finden,

sodass verschiedene Einstellungen der unterschiedlichen Konfigurationsparameter des *Telegraf* Agenten in mehreren Zeit beanspruchenden Testläufen untersucht wurden.

In Bezug auf die Datenaufbereitung in ein „importierfähiges“ Format erweist sich die *InfluxDB* als weitaus flexibler als der *Event Store*, zumal für das Schreiben von Daten mit dem *Telegraf* Agenten in die *InfluxDB* aus einer Vielzahl an Inputformaten gewählt werden kann. Wie unter Kapitel 5.1 beschrieben, werden die Apple Health Daten aus den XML-Dateien im konkreten Anwendungsfall unabhängig von dem DBMS mithilfe eines Skriptes in der Programmiersprache *R* im Zuge einer allgemeinen Datenaufbereitung anonymisiert und liegen als Ergebnis dieser allgemeinen Datenaufbereitung zunächst in einem Dataframe in Tabellenform vor. Da die im Dataframe vorliegenden Apple Health Daten mit der in *R* verfügbaren Funktion „write.csv“ unkompliziert und ohne weiteren Aufwand als CSV-Datei gespeichert werden können, wurde aus dem Angebot der unterstützten Input-Formaten CSV für das Schreiben von Daten in die *InfluxDB* ausgewählt. Auch hinsichtlich des Formats der in der CSV-Datei enthaltenen Daten zeigt sich die *InfluxDB* als unkompliziert, da es mit Ausnahme vom Zeitstempel, der in einem Zeitformat der Go „reference time“ formatiert sein muss, keine weiteren Formatvorgaben zu beachten sind. Die Umwandlung der Zeitwerte in das geforderte Zeitformat kann mit *R* ebenfalls ohne großen Aufwand umgesetzt werden.

Im Gegensatz dazu werden vom *Event Store* nur die beiden Formate JSON und XML für das Schreiben von Daten in das DBMS unterstützt, wobei zusätzliche Funktionalitäten, wie das Schreiben von mehreren Events in einem Batch, sogar nur mit benutzerdefinierten JSON- und XML-Formaten des *Event Stores* genutzt werden können.[189] Die Umwandlung der allgemein aufbereiteten und anonymisierten Apple Health Daten aus dem Dataframe in das für das Schreiben in den *Event Store* gewählte, benutzerdefinierte JSON-Format „application/vnd.eventstore.events+json“ bedarf einiger Zeilen Programmiercode in *R* (s. Anhang D bis F) und ist bei weitem komplexer als die zuvor beschriebene Erstellung von CSV-Dateien mit einer einzigen, in *R* bereits verfügbaren, Funktion. Ein weiterer Aufwand, der beim *Event Store* im Rahmen der Datenaufbereitung und nicht wie bei der *InfluxDB* als Teil der Konfiguration zu berücksichtigen ist, entsteht durch die vorab durchzuführende Definition, wie die Apple Health Daten inhaltlich im Schema des *Event Stores*, bestehend aus Event Stream, Event-Typ, Daten und Metadaten, gespeichert werden sollen.

Zwar ist die Datenaufbereitung beim *Event Store* mit einem gewissen Aufwand verbunden, allerdings gestaltet sich der Schreibprozess der Daten, sobald sie einmal in dem für Batch Writes geeigneten Format „application/vnd.eventstore.events+json“ vorliegen, umso einfacher, da für das Schreiben von Daten in den *Event Store* lediglich ein POST Request gesendet werden muss. Die mit dem POST Request in den *Event*

[189] Vgl. Chinchilla (2018a).

Store geschriebenen Events können anschließend im jeweiligen Event Stream über das Admin UI des *Event Stores* eingesehen werden.

Wie beim *Event Store* offenbart sich das Schreiben von Daten auch bei der *InfluxDB* als sehr einfach, da dafür lediglich der *Telegraf* Agenten zu starten ist. Als Voraussetzung für einen erfolgreichen Schreibprozess ist jedoch eine angemessene Konfiguration zu nennen. Generell ist beim Verwenden des *Telegraf* Agenten aufgefallen, dass dieser eher auf das Schreiben von Streaming Daten ausgelegt ist, da die zu schreibenden Daten in dem in der Konfigurationsdatei angegebenen Intervall (vgl. Kapitel 5.1.1) immer wieder gelesen und geschrieben werden. Da im Rahmen des praktischen Vergleichs allerdings nur eine limitierte Menge an Zeitreihendaten zur Verfügung steht, ist das fortlaufende Sammeln von Input-Daten durch den *Telegraf* Agenten nicht notwendig und daher ggf. nicht die optimale Lösung.

Die Performance beim Schreiben von Daten in die *InfluxDB* bzw. in den *Event Store* weicht bei den im Rahmen des praktischen Vergleichs durchgeführten Schreibvorgängen deutlich voneinander ab. Während bei Batch Größen zwischen 10.000 und 80.000 Metriken je Batch durchschnittlich 27.425 Metriken pro Sekunde in die *InfluxDB* geschrieben werden konnten, betrug die durchschnittliche Anzahl der in den *Event Store* geschriebenen Events bei denselben Batch Größen nur 8.001 Events pro Sekunde, also ca. 71% weniger. Die *InfluxDB* ist beim Schreiben von Daten somit weitaus performanter als der *Event Store*. Als Gemeinsamkeit kann bei den für die Performance-Analyse durchgeführten Schreibvorgängen allerdings festgestellt werden, dass die Batch Größe sowohl bei der *InfluxDB* als auch beim *Event Store* keinen direkten Einfluss auf die Anzahl der durchschnittlich in das DBMS geschriebenen Daten pro Sekunde hat. Außerdem können beim Schreiben von Daten in beide DBMS Schwankungen der Performance beobachtet werden. Bei der *InfluxDB* äußern sich diese durch die weit auseinander liegenden Minimum- und Maximum-Werte der durchschnittlichen Anzahl geschriebener Metriken pro Sekunde je Batch Größe. Beim *Event Store* werden die Schwankungen ebenfalls durch die deutlich variierende durchschnittliche Anzahl geschriebener Events pro Sekunde bei den unterschiedlichen Schreibvorgängen einer Batch Größe und darüber hinaus sogar durch teilweise fehlschlagende Batch Writes ersichtlich. Diese unterschiedliche Performance bei den Schreibvorgängen kann ggf. auf die Tatsache zurückgeführt werden, dass auf dem für den praktischen Vergleich zur Verfügung stehenden Server mehr als 20 virtuelle Maschinen laufen (vgl. 4.1.1). Vor diesem Hintergrund ist nicht auszuschließen, dass unter anderen Hardware Bedingungen sowohl bei der *InfluxDB* als auch beim *Event Store* eine durchschnittlich höhere Anzahl an Metriken bzw. Events in das jeweilige DBMS geschrieben werden kann. Aus diesem Grund sind die im Rahmen des praktischen Vergleichs ermittelten Werte eher als Tendenzen und nicht als genaue Performance-Angaben zu verstehen.

Neben den zuvor beschriebenen Punkten bzgl. der Installation und Konfiguration, der Datenaufbereitung sowie des Schreibvorgangs konnten sowohl bei der *InfluxDB* als auch beim *Event Store* Besonderheiten identifiziert werden. Bei der *InfluxDB* ist zu be-

achten, dass es nicht möglich ist, mehrere unterschiedliche Messungen in das DBMS zu schreiben, die den gleichen timestamp sowie die gleichen Tag Werte besitzen. Ist bereits eine Messung einer bestimmten Aktivität zu einem bestimmten Zeitstempel und mit bestimmten Tags in der *InfluxDB* gespeichert, kann folglich weder eine zweite Messung dieser Aktivität mit dem gleichen timestamp sowie den gleichen Tags in die *InfluxDB* geschrieben werden, noch werden die vorhandenen, bereits gespeicherten Daten überschrieben. Da in den vorliegenden Apple Health Daten aber vereinzelt mehrere Messungen der gleichen Aktivität zum gleichen Zeitstempel und mit den gleichen Tags enthalten sind, kommt es dazu, dass diese Messungen nicht in die *InfluxDB* geschrieben werden. Die Differenz zwischen der Anzahl an Messwerten in den Originaldaten und der in die *InfluxDB* geschriebenen Messwerte beträgt dabei 10.350 Messungen, was bei den insgesamt 3.264.383 vorliegenden Messwerten aus den Apple Health Daten nur einen Anteil von 0,32% ausmacht. Um dem Fehlen von einzelnen Zeitreihendaten, wie es im Rahmen des praktischen Vergleich vorkommt, vorzubeugen, sollte bei der Verwendung der TSDB *InfluxDB* darauf geachtet werden, dass die Daten mit einem Zeitstempel in einer möglichst kleinen Zeiteinheit erfasst werden oder dass das Erfassen von Daten bestimmter Messungen mit den gleichen Tags zum gleichen Zeitstempel im Quellsystem technisch nicht möglich ist.

Beim *Event Store* besteht die Besonderheit darin, dass die Daten bzw. Events immer den timestamp des Zeitpunkts erhalten, zu dem sie in den *Event Store* geschrieben werden. Zusätzlich zu diesem Zeitstempel, wird für jedes Event innerhalb eines Event Streams eine fortlaufende Nummer vergeben, welche die zeitliche Reihenfolge der Events widerspiegelt. Im Rahmen des praktischen Vergleichs mit den Apple Health Daten ist es somit nicht möglich, den Zeitpunkt des „endDate“s der Aktivitätsmessungen als Zeitstempel im *Event Store* zu definieren. Aus diesem Grund ist das jeweilige „endDate“ in den Daten („data“) eines jeden einzelnen Events enthalten.

5.2 Abfragen

Der zweite Teil des praktischen Vergleichs der ausgewählten TSDB und des ausgewählten Event Stores behandelt Abfragen der zuvor in die DBMS geschriebenen Daten. Der Fokus liegt hierbei auf den qualitativen bzw. weichen Aspekten, wie Komplexität und Usability. In beiden DBMS sollen jeweils drei verschiedene Abfragen durchgeführt werden, die im Folgenden beschrieben werden.

(1) Zeitspezifische Abfrage

Bei der ersten Abfrage soll als Bedingung lediglich der Faktor Zeit eingeschränkt werden. Im konkreten Anwendungsfall der Apple Health Daten sollen somit beispielhaft alle Werte einer ausgewählten Aktivität für einen bestimmten Zeitraum abgefragt werden. Für die erste Abfrage im Rahmen des praktischen Vergleichs werden hierfür die in der Tabelle 5-4 zu sehenden Anforderungen definiert:

Objekt	Bedingung
Aktivitätsmessung	„StepCount“
Zeit	>= 07.01.2019 00:00 Uhr und < 08.01.2019 00:00 Uhr

Tabelle 5-4: Bedingungen der Abfrage 1

(2) Kombination zeitspezifische und nicht-zeitspezifische Abfrage

Die zweite Abfrage soll zusätzlich zum Faktor Zeit eine weitere Bedingung beinhalten. Im vorliegenden Anwendungsfall der Apple Health Daten sollen folglich, wie bei der ersten Abfrage, alle Werte einer ausgewählten Aktivität für einen bestimmten Zeitraum mit der zusätzlichen Einschränkung auf ein weiteres, nicht-zeitspezifisches Kriterium, wie bspw. einen User oder ein Gerät, abgefragt werden. Die Tabelle 5-5 zeigt die Bedingungen für die zweite im Rahmen des praktischen Vergleichs durchzuführende Abfrage.

Objekt	Bedingung
Aktivitätsmessung	„FlightsClimbed“
Zeit	>= 01.09.2018 00:00 Uhr und < 01.11.2018 00:00 Uhr
„userDeviceSourceID“	„User1_iPhone_2“

Tabelle 5-5: Bedingungen der Abfrage 2

(3) Kombination zeitspezifische und nicht-zeitspezifische Abfrage inklusive Aggregation

Bei der dritten Abfrage soll zusätzlich zu den Bedingungen auf ein zeitspezifisches und ein nicht-zeitspezifisches Objekt eine Aggregation der Werte nach einem zeitlichen und in einem zweiten Schritt auch nach einem nicht-zeitlichen Faktor stattfinden. Als Ergebnis der Abfrage sollen demnach nicht die einzelnen in den beiden DBMS gespeicherten Werte für den abgefragten Zeitraum, sondern bereits aggregierte Werte wiedergegeben werden. Im Rahmen des praktischen Vergleichs soll die Aggregation in Form einer Summe der Werte für das Zeitintervall von jeweils sieben Tagen und im zweiten Schritt aggregiert nach der „userID“ erfolgen. Die weiteren Bedingungen für die dritte durchzuführende Abfrage sind in der Tabelle 5-6 enthalten.

Objekt	Bedingung
Aktivitätsmessung	„StepCount“
Zeit	>= 01.01.2018 00:00 Uhr und < 01.01.2019 00:00 Uhr
„deviceAnonymous“	„iPhone“

Tabelle 5-6: Bedingungen der Abfrage 3

5.2.1 InfluxDB

Wie in Kapitel 4.2.1 erläutert, können die in der *InfluxDB* gespeicherten Daten mit InfluxQL, einer SQL-ähnlichen Abfragesprache, abgefragt werden. In der online verfügbaren Dokumentation der *InfluxDB* ist ebenfalls die InfluxQL ausführlich dokumentiert.[190] Zum einen ist es möglich, die Abfragen über das Command Line Interface (CLI) einzugeben, welches über den Befehl „`influx`“ inkl. des Vermerks auf den richtigen Port („`-port '8082'`“) gestartet werden kann. Hier ist es möglich mit dem Befehl „`USE <database>`“ eine in der *InfluxDB* vorhandene Datenbank auszuwählen, die für alle folgenden Abfragen verwendet wird.[191] Für den konkreten Anwendungsfall wird mit dem Befehl „`USE „thesisKS“`“ die Datenbank „thesisKS“ ausgewählt, in welche die gesamten vorliegenden Apple Health Daten geschrieben wurden. Zum anderen können die Abfragen über das UI *Chronograf* mit einer Art Abfrageassistent erstellt und mit unterschiedlichen Visualisierungstypen, wie Tabellen und Diagramme, veranschaulicht werden.[192] Beide Möglichkeiten der Datenabfrage werden im Rahmen des praktischen Vergleichs getestet.

(1) Zeitspezifische Abfrage

Die erste zeitspezifische Abfrage mit den in der Tabelle 5-4 aufgeführten Bedingungen kann mit folgender Abfrage in der InfluxQL über das Command Line Interface (CLI) umgesetzt werden.

```
SELECT "value" FROM "StepCount" WHERE time >= '2019-01-07T00:00:00Z' AND time < '2019-01-08T00:00:00Z'
```

Als Ergebnis dieser Abfrage werden die Werte der Messung „StepCount“ für die Zeitstempel innerhalb des in der Abfrage definierten Zeitintervalls in Form einer Liste wiedergegeben, wie der Ausschnitt des Abfrageergebnisses in der Abbildung 5-11 zeigt.

```
InfluxDB shell version: 1.7.7
> USE "thesisKS"
Using database thesisKS
> SELECT "value" FROM "StepCount" WHERE time >= '2019-01-07T00:00:00Z' AND time
< '2019-01-08T00:00:00Z'
name: StepCount
time                value
----                -----
1546823375000000000 2
1546827406000000000 82
1546827971000000000 15
1546827978000000000 67
1546828737000000000 102
```

Abbildung 5-11: Ausschnitt des Ergebnisses der Abfrage 1 im InfluxDB CLI

[190] Vgl. Bang (2018a).

[191] Vgl. Bang (2019b).

[192] Vgl. InfluxData (2019a).

Im UI *Chronograf* können die Abfragen mit der InfluxQL genau wie im CLI durchgeführt werden, mit dem einzigen Unterschied, dass bei den Abfragen im UI *Chronograf* die Datenbank innerhalb der Abfrage angegeben werden muss, wie in der folgenden im UI *Chronograf* eingegeben Abfrage fett hervorgehoben wird.

```
SELECT "value" FROM "thesisKS"."autogen"."StepCount" WHERE time
>= '2019-01-07T00:00:00Z' AND time < '2019-01-08T00:00:00Z'
```

Um das Ergebnis der Abfrage im UI *Chronograf* mit dem Ergebnis aus dem CLI zu vergleichen, wird zunächst eine Tabelle als Visualisierungstyp ausgewählt. Die Abbildung 5-12 zeigt den gleichen Ausschnitt des Abfrageergebnisses im UI *Chronograf* wie die Abbildung 5-11 aus dem CLI. Ein großer Vorteil, den *Chronograf* gegenüber dem CLI bietet, ist, dass der Zeitstempel im UI *Chronograf* in einem lesbaren Datums- und Uhrzeit-Format dargestellt wird, während der Zeitstempel im Abfrageergebnis im CLI im UNIX-timestamp mit der kleinsten Zeiteinheit Mikrosekunde angegeben wird und daher für den Anwender oder die Anwenderin nicht direkt verständlich ist. Ein weiterer Vorteil des UI *Chronograf* ist die Möglichkeit, das Abfrageergebnis als CSV-Datei herunterzuladen, wie ebenfalls in der Abbildung 5-12 erkannt werden kann. Diese Funktionalität eignet sich für Anwendungsfälle, wo die Zeitreihendaten bspw. mit Daten aus externen Systemen in Verbindung gebracht oder in einer anderen Weise weiterverarbeitet werden.

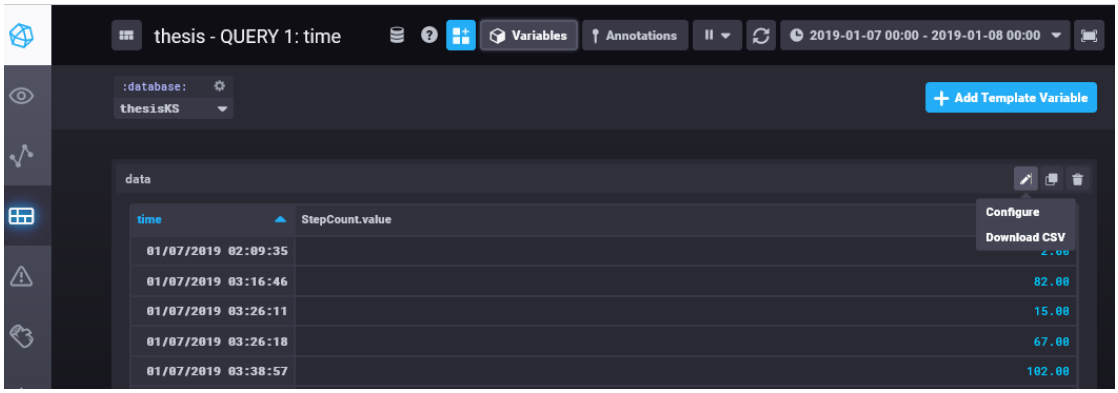

Abbildung 5-12: Ausschnitt des Ergebnisses der Abfrage 1 in Tabellenform im UI Chronograf

Mithilfe des Exports als CSV-Datei konnte eine Diskrepanz zwischen den über das UI *Chronograf* abgefragten Daten und den über das CLI abgefragten Daten festgestellt werden. Während das Abfrageergebnis über das CLI 1.097 Werte der Messung „StepCount“ beinhaltet, besteht das über *Chronograf* abgefragte Ergebnis lediglich aus 1.089 Werten. Durch eine detaillierte Betrachtung der beiden Abfrageergebnisse konnte der Grund für die Diskrepanz lokalisiert werden. Er liegt darin, dass im Abfrageergebnis über das UI *Chronograf* lediglich ein Wert je Zeitstempel enthalten ist, während im Abfrageergebnis über das CLI in insgesamt 15 Fällen bzw. bei sieben verschiedenen Zeitstempeln mehrere Werte je Zeitstempel wiedergegeben werden. Dies ist inhaltlich plausibel, da mit dieser ersten Abfrage die Daten aller 27 User abgefragt wurden und bei der Betrachtung unterschiedlicher User mehrere Werte einer Messung für ei-

nen einzigen Zeitstempel vorliegen können. Unter Berücksichtigung dieser Erkenntnis sollten Abfragen unbedingt über das CLI ausgeführt werden, sofern die Möglichkeit besteht, dass mehrere Werte für einen Zeitstempel vorliegen und ein vollständiges Abfrageergebnis erwartet wird.

Nichtsdestotrotz eignet sich das UI *Chronograf* für eine einfache Visualisierung der Daten. Neben der Darstellung in Tabellenform in der Abbildung 5-12 kann das Ergebnis der ersten Abfrage durch eine unkomplizierte Änderung des Visualisierungstyps im UI bspw. auch in einem Liniendiagramm dargestellt werden, wie in der Abbildung 5-13 zu sehen ist. Diese Darstellungsform erleichtert die Interpretation der Daten. Im Gegensatz zur Darstellung der Daten in einer Tabelle kann mithilfe des Diagramms sofort erkannt werden, dass vor 06:00 Uhr und ab 23:00 Uhr kaum Schritte („StepCount“) gemessen wurden, was auf die Schlafenszeit der User zurückzuführen ist.

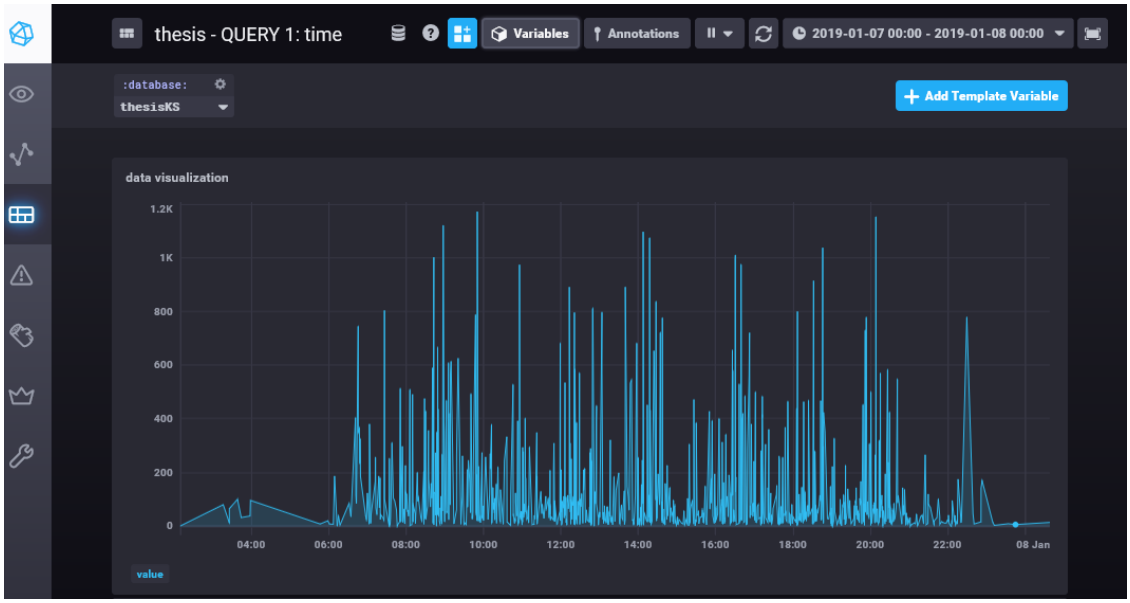

Abbildung 5-13: Visualisierung des Ergebnisses der Abfrage 1 im UI Chronograf

(2) Kombination zeitspezifische und nicht-zeitspezifische Abfrage

Die zweite Abfrage, welche die in der Tabelle 5-5 dargestellten zeitspezifischen und nicht-zeitspezifischen Bedingungen berücksichtigen soll, kann mit folgender InfluxQL-Abfrage über das CLI erfolgen.

```
SELECT "value" FROM "FlightsClimbed" WHERE "userDeviceSourceID"='User1_iPhone_2' AND time >= '2018-09-01T00:00:00Z' AND time < '2018-11-01T00:00:00Z'
```

Wie bei der ersten Abfrage wird auch das Ergebnis der zweiten Abfrage, welches aus insgesamt 383 Werten der Messung „FlightsClimbed“ mit der „userDeviceSourceID“ „User1_iPhone_2“ für die Zeitstempel innerhalb des abgefragten Zeitintervalls besteht, im CLI in Form einer Liste wiedergegeben, wie in der Abbildung 5-14 zu sehen ist.

```
> SELECT "value" FROM "FlightsClimbed" WHERE "userDeviceSourceID"='User1_iPhone_
2' AND time >= '2018-09-01T00:00:00Z' AND time < '2018-11-01T00:00:00Z'

name: FlightsClimbed
time                value
----                -----
1535786775000000000 1
1535787704000000000 1
1535787727000000000 1
1535805974000000000 1
1535831704000000000 1
1535885796000000000 1
```

Abbildung 5-14: Ausschnitt des Ergebnisses der Abfrage 2 im InfluxDB CLI

Mit folgender InfluxQL-Abfrage unter der Angabe der Datenbank innerhalb der *InfluxDB* kann die zweite Abfrage im UI *Chronograf* umgesetzt werden.

```
SELECT "value" FROM "thesisKS"."autogen"."FlightsClimbed" WHERE
"userDeviceSourceID"='User1_iPhone_2' AND time >= '2018-09-
01T00:00:00Z' AND time < '2018-11-01T00:00:00Z'
```

Da diese Abfrage nur die Werte der „userDeviceSourceID“ „User1_iPhone_2“ und somit nur die Werte eines einzigen Users abfragt, liegen nicht mehrere Werte für einen einzigen Zeitstempel vor, sodass auch das Abfrageergebnis über *Chronograf* aus 383 Werten der Messung „FlightsClimbed“ besteht.

Zumal eine Liste 383 einzelner Werte nur schwer analysierbar ist, wird für die zweite Abfrage ein Balkendiagramm als Visualisierungstyp im UI *Chronograf* ausgewählt. Dank der Veranschaulichung kann schnell erkannt werden, dass die Anzahl der erklommenen Stockwerke im Zeitraum von Ende September bis Anfang Oktober deutlich höher ist, als zu den anderen abgefragten und visualisierten Zeitpunkten.

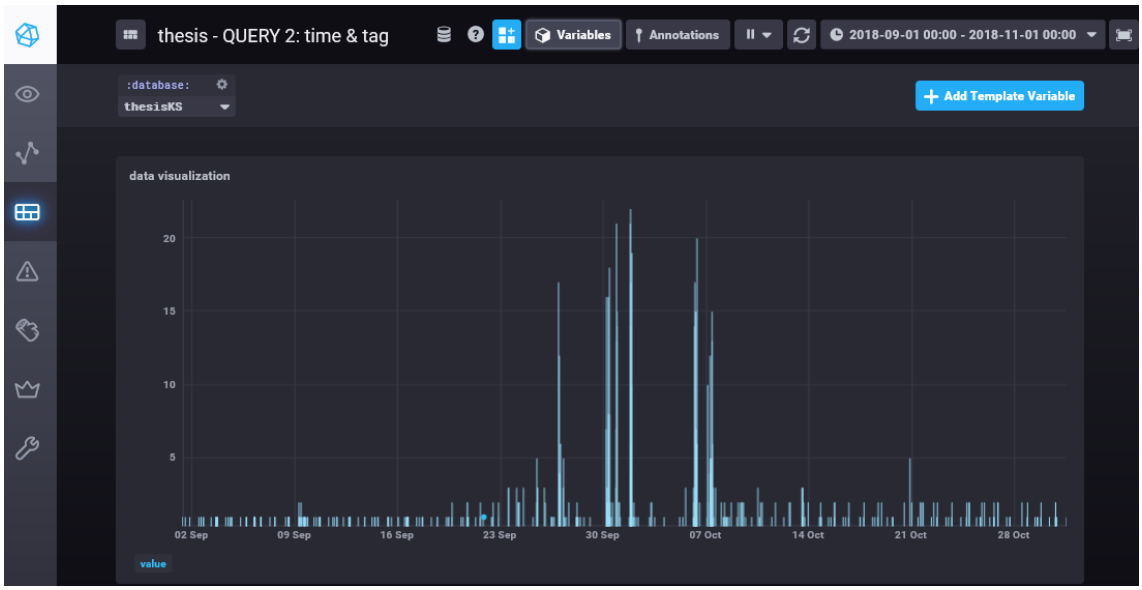

Abbildung 5-15: Visualisierung des Ergebnisses der Abfrage 2 im UI Chronograf

(3) Kombination zeitspezifische und nicht-zeitspezifische Abfrage inklusive Aggregation

Die dritte im Rahmen des praktischen Vergleichs durchzuführende Abfrage soll zusätzlich zu den in der Tabelle 5-6 aufgeführten Bedingungen eine Aggregation in Form einer Summe der Werte für das Zeitintervall von jeweils sieben Tagen enthalten. In der InfluxQL lässt sich diese Anforderung mit der Funktion „SUM()“ und einer „GROUP BY time ()“-Bedingung, welche das Zeitintervall beinhaltet, umsetzen, wie auch in der folgenden über das CLI getätigten Abfrage in der InfluxQL zu sehen ist.[193]

```
SELECT sum("value") FROM "StepCount" WHERE time >= '2018-01-01T00:00:00Z' AND time < '2019-01-01T00:00:00Z'AND "deviceAnonymous"='iPhone' GROUP BY time(7d)
```

Um die Daten über das UI *Chronograf* abzufragen, ist wie bei den ersten beiden Abfragen lediglich der Name der Datenbank (`"thesisKS"."autogen"`) zu ergänzen. Unabhängig vom Abfrage Interface besteht das Abfrageergebnis der zuvor aufgeführten Abfrage aus 53 über alle User hinweg summierten Werte der auf den iPhones der User gemessenen zurückgelegten Schritte („StepCount“), aggregiert auf sieben Tage.

Für die Aggregation nach dem nicht-zeitspezifischem Objekt „userID“ im zweiten Schritt der dritten Abfrage kann einfach eine weitere „GROUP BY“-Bedingung ergänzt werden, wie in der folgenden Abfrage fett markiert ist. Hierbei ist zu beachten, dass lediglich die beim Schreiben der Daten in die *InfluxDB* definierten Tags und keine Fields (vgl. Kapitel 5.1.1) als „GROUP BY“-Bedingung verwendet werden können.[194]

```
SELECT sum("value") FROM "StepCount" WHERE time >= '2018-01-01T00:00:00Z' AND time < '2019-01-01T00:00:00Z'AND "deviceAnonymous"='iPhone' GROUP BY time(7d), "userID"
```

Als Ergebnis dieser Abfrage werden für jeden der 27 User 53 auf sieben Tage aggregierte bzw. summierte Werte der Messung „StepCount“, also in Summe 1.431 Werte, wiedergegeben. Der Ausschnitt des Abfrageergebnisses in der Abbildung 5-16 zeigt die ersten acht aggregierten Werte für den „User1“. Durch diese Abbildung soll die Gliederung des Abfrageergebnisses nach den unterschiedlichen „userID“s als Tags verdeutlicht werden.

[193] Vgl. Bang/Anderson (2019c).

[194] Vgl. Bang/Anderson (2019b).

```
> SELECT sum("value") FROM "StepCount" WHERE time >= '2018-01-01T00:00:00Z' AND
time < '2019-01-01T00:00:00Z'AND "deviceAnonymous"='iPhone' GROUP BY time(7d), "
userID"
name: StepCount
tags: userID=User1
time                sum
----                ---
1514419200000000000 16835
1515024000000000000 26985
1515628800000000000 26822
1516233600000000000 19444
1516838400000000000 29870
1517443200000000000 20848
1518048000000000000 68782
1518652800000000000 36094
```

Abbildung 5-16: Ausschnitt des Ergebnisses der Abfrage 3 im InfluxDB CLI

Für die Veranschaulichung der Daten mithilfe des UI *Chronograf* wird für die Aggregationsabfrage über alle User hinweg ein Balkendiagramm und für die Aggregation je User ein Liniendiagramm verwendet, wie in der Abbildung 5-17 zu sehen ist. Durch die Visualisierung der Werte mehrerer User in einem Liniendiagramm, bei dem die Zeit auf der X-Achse dargestellt ist, können die Unterschiede zwischen den Usern zu bestimmten Zeitpunkten im Gegensatz zu den zuvor beschriebenen über das CLI ausgegebenen Abfrageergebnissen in Form einer Liste je User schneller erkannt werden. Für detailliertere Betrachtungen ist darüber hinaus ein Drill-Down bzw. Zoom durch die Interaktion mit dem Diagramm möglich.

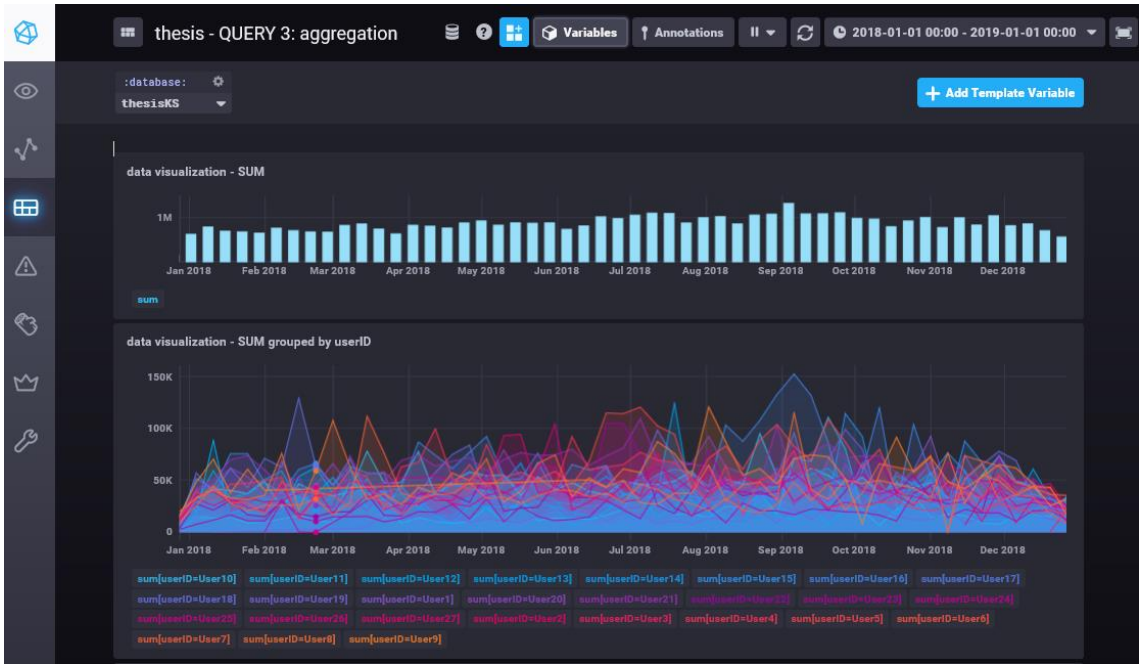

Abbildung 5-17: Visualisierung des Ergebnisses der Abfrage 3 im UI Chronograf

5.2.2 Event Store

Die in den *Event Store* geschriebenen Daten können zum einen über GET Requests gelesen werden. In diesen Anfragen ist der Event Stream anzugeben, von welchem die Events gelesen werden sollen, sowie die Nummer des zu lesenden Events innerhalb des Event Streams. Mit der folgenden Abfrage wird bspw. das erste Event, welches in einem Event Stream immer die Nummer Null hat, im Event Stream „User1thesisKS50k“ gelesen.

```
curl -i http://139.6.56.162:2113/streams/User1thesisKS50k/0 - -H
"Accept: application/json"
```

Die Antwort auf diesen GET Request kann in der Abbildung 5-18 betrachtet werden. Sie beinhaltet die in den Daten („data“) des Events gespeicherten Werte.

```
schinker@lwivs32:~$ sudo curl -i http://139.6.56.162:2113/streams/UserlthesisKS5
0k/0 -u admin:changeit -H "Accept: application/json"
HTTP/1.1 200 OK
Access-Control-Allow-Methods: GET, OPTIONS
Access-Control-Allow-Headers: Content-Type, X-Requested-With, X-Forwarded-Host,
X-Forwarded-Prefix, X-PINGOTHER, Authorization, ES-LongPoll, ES-ExpectedVersion,
 ES-EventId, ES-EventType, ES-RequiresMaster, ES-HardDelete, ES-ResolveLinkTos
Access-Control-Allow-Origin: *
Access-Control-Expose-Headers: Location, ES-Position, ES-CurrentVersion
Cache-Control: max-age=31536000, public
Vary: Accept
Content-Type: application/json; charset=utf-8
Server: Mono-HTTPAPI/1.0
Date: Tue, 30 Jul 2019 19:53:29 GMT
Content-Length: 164
Keep-Alive: timeout=15,max=100

{
  "value": 11,
  "startDate": "2016-11-21 15:02:45",
  "endDate": "2016-11-21 15:10:38",
  "deviceAnonymous": "iPhone",
  "userDeviceSourceID": "User1_iPhone_2"
```

Abbildung 5-18: Antwort auf einen GET Request zum Lesen von Events aus Event Streams

Zum anderen können die in den *Event Store* geschriebenen Daten über das Admin UI im Reiter „Stream Browser“ gelesen werden. Auch hier kann wie im GET Request ein Event Stream sowie ein einzelnes Event mit seiner Nummer innerhalb des Event Streams ausgewählt werden, von welchem die Daten („data“) und Metadaten („metadata“) angezeigt werden können. In der Abbildung 5-19 wird das gleiche Event, wie mit dem zuvor genannten GET Request abgefragt und in der Abbildung 5-18 veranschaulicht, im Admin UI gelesen.

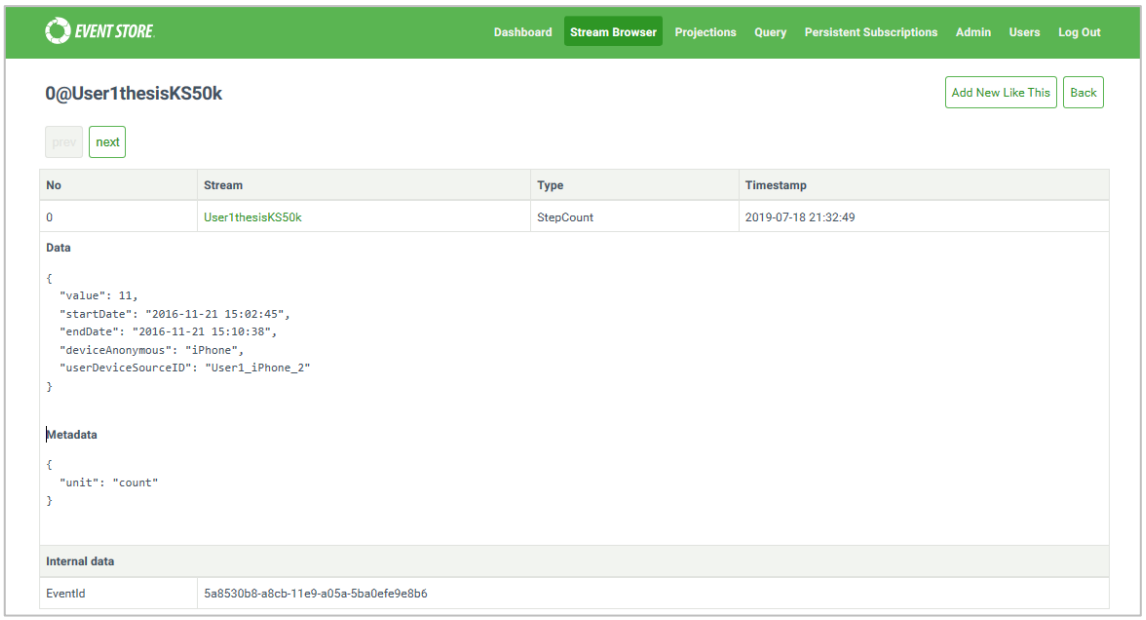

Abbildung 5-19: Lesen von Events aus Event Streams im Admin UI

Beide dieser Varianten zum Lesen von Daten aus dem *Event Store* eignen sich jedoch nicht, um die drei im Rahmen des praktischen Vergleichs durchzuführenden Abfragen umzusetzen. Zwar kann sowohl in den GET Requests als auch im Admin UI aktiv ein Event Stream und die Position eines Events innerhalb des Event Streams ausgewählt werden, jedoch ist keine Abfrage eines bestimmten Zeitpunktes bzw. -intervalls mithilfe des Zeitstempels oder die Einschränkung auf bestimmte in den Daten („data“) des Events gespeicherten Kriterien möglich. Da der timestamp eines im *Event Store* gespeicherten Events dem Zeitpunkt des Schreibens der Daten in den *Event Store* entspricht und der eigentliche Zeitstempel der Apple Health Daten im vorliegenden Anwendungsfall als „endDate“ in den Daten („data“) eines Events gespeichert ist (vgl. Kapitel 5.1.2), ist bei den drei durchzuführenden Abfragen zwingend notwendig, dass eine Einschränkungen bzw. Filterung auf die Daten („data“) eines Events möglich ist.

Eine weitere Möglichkeit zum Abfragen von Events im *Event Store* sind sogenannte Projections. Dabei handelt es sich um ein Subsystem des *Event Stores*, mit dem neue Events geschrieben oder bestehende Events reaktiv zu Event Streams verknüpft werden können. Projections sind insbesondere für zeitliche Korrelationsabfragen geeignet.[195] Bezugnehmend auf die Apple Health Daten ist ein Beispiel für eine zeitliche Korrelationsabfrage, ob eine steigende bzw. hohe Herzfrequenz („HeartRate“) zusammen mit einer steigenden bzw. hohen Anzahl an zurückgelegten Schritten („StepsCount“) je Zeiteinheit auftritt. Mit einer Projection kann bspw. ein bei Null beginnender Zähler gestartet werden, der die Anzahl der Fälle zählt, wo die beschriebene Korrelation vorliegt. Der aktuelle Zustand einer Projection kann ebenfalls abgefragt werden.

[195] Vgl. Chinchilla (2019c).

Für die Aktivierung von Projections im *Event Store* ist die Konfiguration von zwei weiteren Paramatern, nämlich „RunProjections: All“ und „ StartStandardProjections: True“, in der Konfigurationsdatei „eventstore.conf“ notwendig, wie in der Abbildung 5-20 gezeigt wird.[196] Nach erfolgreicher Aktivierung, können benutzerdefinierte Projections in JavaScript über den Reiter „Projections“ im Admin UI geschrieben werden.[197]

```
---
IntIp: 139.6.56.162
ExtIp: 139.6.56.162

RunProjections: All
StartStandardProjections: True
---
```

Abbildung 5-20: Inhalt der Konfigurationsdatei „eventstore.conf“ inklusive der Einstellungen zu den Projections

Zwar bestehen Projections auch aus einem Selektor, der Events von einem oder mehreren angegebenen Event Streams auswählt, sowie Filter- bzw. Partitionsmöglichkeiten, die nur jene Events die Projections passieren lassen, welche die angegebenen Anforderungen erfüllen, sodass die drei im Rahmen des praktischen Vergleichs durchzuführenden Abfragen mit Projections voraussichtlich umgesetzt werden könnten.[198] Allerdings werden Projections als sehr komplex und daher mit einem hohen Aufwand verbunden eingestuft. Aus diesem Grund werden Projections für die Art von klassischen Abfragen von Daten, wie sie für den praktischen Vergleich durchzuführen sind, als weniger geeignet betrachtet und somit im Folgenden nicht weiter forciert.

5.2.3 Vergleich der InfluxDB und des Event Stores bei Abfragen

Bei der *InfluxDB* konnten die drei für den praktischen Vergleich definierten Abfragen ohne Einschränkung durchgeführt werden. Die SQL-ähnliche Abfragesprache InfluxQL ist auch für Anwender und Anwenderinnen mit nur rudimentären SQL-Kenntnissen leicht verständlich, sodass die drei durchzuführenden Abfragen, aber auch zusätzliche Abfragen in diesem Anwendungsfall mit wenig Aufwand umgesetzt werden konnten. Zusätzlich bieten die unterschiedlichen Abfrage Interfaces eine Flexibilität und insbesondere das UI *Chronograf* eine gute Usability für den Anwender oder die Anwenderin. Im vorliegenden Anwendungsfall zeigt sich die *InfluxDB* somit als sehr geeignet.

Nichtsdestotrotz konnte bei der *InfluxDB* auch ein Optimierungspotential für das Erstellen von Abfragen identifiziert werden. Bei Aggregationen können aktuell nämlich nur die Zeiträume bzw. -einheiten Nanosekunde, Mikrosekunde, Millisekunde, Sekunde,

[196] Vgl. Chinchilla (2019c).

[197] Vgl. Chinchilla (2018c).

[198] Vgl. Chinchilla (2018c).

Minute, Stunde, Tag und Woche als „GROUP BY time()“-Bedingung verwendet werden. Ein „GROUP BY time ()“ nach Monaten oder Jahren ist somit nicht möglich. Zwar war diese Funktionalität für die drei im Rahmen des praktischen Vergleichs durchzuführenden Abfragen nicht erforderlich, jedoch können die Zeiteinheiten Monat und Jahr bei Aggregationen in anderen Anwendungsfällen sicherlich von Vorteil sein.

Im Gegensatz dazu hat sich der *Event Store* für die Durchführung der drei für den praktischen Vergleich definierten Abfragen als weniger zweckmäßig herausgestellt, weshalb die Abfragen auch nicht durchgeführt wurden. Während einfache GET Requests und das Admin UI lediglich für das Anzeigen bzw. das Lesen von einzelnen Events innerhalb eines Event Streams verwendet werden können, sind umfangreichere Abfragen der im *Event Store* gespeicherten Daten nur mit Projections möglich. Da sich diese jedoch weniger für klassische Datenabfragen, sondern vielmehr für die Abfrage zeitlicher Korrelationen eignen und die in JavaScript zu schreibenden benutzerdefinierten Projections darüber hinaus eine hohe Komplexität aufweisen, wird von der Verwendung von Projections im Rahmen des praktischen Vergleichs abgesehen.

6 Fazit

Als erster Teil des Vergleichs wurden TSDBs und Event Stores auf theoretischer Ebene anhand funktionaler und nicht-funktionaler Kriterien verglichen. Aus diesem theoretischen Vergleich geht hervor, dass sowohl TSDBs als auch Event Stores für die in der Einleitung beschriebenen steigenden Datenmengen geeignet sind und auch den daraus folgenden steigenden Anforderungen an die Datenspeicherung und -analyse gerecht werden. Bei beiden Arten von DBMS sind jedoch Unterschiede zwischen einzelnen TSDBs bzw. einzelnen Event Stores zu erkennen, sodass stets im konkreten Anwendungsfall, unter Berücksichtigung der gegebenen Anforderungen in Bezug auf die Performance, Skalierbarkeit, verfügbaren Funktionen, angebotenen Support usw., zu prüfen ist, welche TSDB bzw. welcher Event Store am besten geeignet ist.

Um TSDBs und Event Stores im zweiten Teil des Vergleichs auf praktischer Ebene, d.h. bei der Implementierung und bei der Durchführung von Abfragen, miteinander zu vergleichen, wurde aus einer Auswahl an TSDBs und Event Stores jeweils die für den in Kapitel 4 beschriebenen konkreten Anwendungsfall geeignetste TSDB, in diesem Fall *InfluxDB*, und der geeignetste Event Store, nämlich *Event Store*, ausgewählt. Da die für den praktischen Vergleich verwendeten Apple Health Daten im Zeitablauf gemessene Aktivitätsdaten von iPhone- und Apple Watch-Usern und somit klassische Zeitreihendaten darstellen, konnte damit geprüft werden, ob sich TSDBs und Event Stores gleichermaßen für die Speicherung und in einem zweiten Schritt auch für die Abfrage solcher Zeitreihendaten eignen.

Wie zu erwarten erweist sich die TSDB *InfluxDB* im konkreten Anwendungsfall als adäquate Wahl für die Speicherung und Abfrage der Zeitreihendaten, zumal die unterschiedlichen Messungen aus den Aktivitätsdaten der Apple Health App als „measurements“ in der *InfluxDB* gespeichert werden und sowohl zeitspezifische als auch nicht-zeitspezifische Abfragen sowie Aggregationen der Messwerte unkompliziert mithilfe der verfügbaren Funktionen durchgeführt werden können. Mit Bezug auf die in der Einleitung geschilderte Tatsache, dass Zeitreihendaten im Kontext des IoT kontinuierlich von Sensoren generiert werden, gestaltet sich der auf das Schreiben von Streaming Daten ausgelegte *Telegraf* Agent, mit dem Daten in die *InfluxDB* geschrieben werden können, darüber hinaus als besonders nützlich. Dies unterstreicht die Spezialisierung und folglich auch Eignung der *InfluxDB* bzw. der Komponenten der *InfluxData* Plattform für die Speicherung und Abfrage von, ggf. durch Sensoren generierten, Zeitreihendaten.

Im Gegensatz dazu ist der Event Store *Event Store* im konkreten Anwendungsfall zunächst beim Schreiben der Daten in das DBMS weniger performant als die *InfluxDB* und zeigt sich insbesondere bei der Abfrage der Zeitreihendaten als deutlich ungeeigneter. Dies kann darauf zurückgeführt werden, dass Event Stores auf die Speicherung von zustandsverändernden Events spezialisiert sind, deren Wiedergabe in der Reihenfolge ihres Eintreffens den aktuellen Zustand eines Objekts zeigt. Die Aktivitätsdaten der Apple Health App stellen hingegen keine Zustandsveränderungen, sondern vielmehr einzelne, unabhängige Messwerte unterschiedlicher Aktivitäten dar, die nicht nur

als zeitliche Abfolge, sondern auch einzeln, in ausgewählten Teilen oder aggregiert betrachtet werden können. Die Art der im praktischen Vergleich verwendeten Zeitreihendaten ist für die Speicherung in einem Event Store somit nicht optimal.

Als Ergebnis des praktischen Vergleichs kann schließlich festgehalten werden, dass die Zeitreihendaten im konkreten Anwendungsfall zwar in beiden Arten von DBMS gespeichert werden können, die Nutzung der auf Time Series Daten spezialisierten TSDB *InfluxDB* aber offensichtliche Vorteile gegenüber dem Event Store *Event Store* aufweist und somit für die Speicherung und Abfrage von Zeitreihendaten deutlich besser geeignet ist. Die Frage, ob sich TSDBs und Event Stores gleichermaßen für die Speicherung und Abfrage von Zeitreihendaten eignen, kann somit verneint werden, da die auf die Speicherung von Events spezialisierten Event Stores, obwohl Events auch Zeitreihendaten darstellen, nicht für jede Art von Time Series Daten zweckmäßig sind.

Literaturverzeichnis

Anderson, Scott (2019): Installing Chronograf, URL: https://docs.InfluxData.com/Chronograf/v1.7/introduction/installation/, Stand: 1. Juni 2019.

Apple (2019a): A bold way to look at your health., URL: https://www.apple.com/ios/health/, Stand: 19. April 2019.

Apple (2019b): Data Types, URL: https://developer.apple.com/documentation/healthkit/data_types, Stand: 26. April 2019.

Bader, Andreas (2016): Comparison of Time Series Databases, Stuttgart.

Bader, Andreas/Kopp, Oliver/Falkenthal, Michael (2017): Survey and Comparison of Open Source Time Series Databases, in: Mitschang, Bernhard u. a. (Hrsg.): Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband. 6.-10. März 2017 Stuttgart, Bonn, S. 249–268.

Bang, Steve (2018a): Influx Query Language (InfluxQL), URL: https://docs.InfluxData.com/InfluxDB/v1.7/query_language/, Stand: 1. Mai 2019.

Bang, Steve (2018b): InfluxDB 1.7 documentation, URL: https://docs.InfluxData.com/InfluxDB/v1.7/, Stand: 5. Mai 2019.

Bang, Steve (2019a): Downsampling and data retention, URL: https://docs.InfluxData.com/InfluxDB/v1.7/guides/downsampling_and_retention/, Stand: 1. Mai 2019.

Bang, Steve (2019b): Getting started with InfluxDB OSS, URL: https://docs.InfluxData.com/InfluxDB/v1.7/introduction/getting-started/, Stand: 28. Juli 2019.

Bang, Steve (2019c): InfluxDB glossary of terms, URL: https://docs.InfluxData.com/InfluxDB/v1.7/concepts/glossary/#field, Stand: 25. Mai 2019.

Bang, Steve (2019d): Introducing Telegraf, URL: https://docs.InfluxData.com/Telegraf/v1.10/introduction/, Stand: 22. Mai 2019.

Bang, Steve (2019e): Telegraf input data formats, URL: https://docs.InfluxData.com/Telegraf/v1.10/data_formats/input/.

Bang, Steve/Anderson, Scott (2018): InfluxQL Continuous Queries, URL: https://docs.InfluxData.com/InfluxDB/v1.7/query_language/continuous_queries/#continuous-query-use-cases, Stand: 26. Januar 2019.

Bang, Steve/Anderson, Scott (2019a): Configuring InfluxDB OSS, URL: https://docs.InfluxData.com/InfluxDB/v1.7/administration/config/#using-the-configuration-file, Stand: 27. Juni 2019.

Bang, Steve/Anderson, Scott (2019b): Data exploration using InfluxQL, URL: https://docs.InfluxData.com/InfluxDB/v1.7/query_language/data_exploration/#the-group-by-clause, Stand: 29. Juli 2019.

Bang, Steve/Anderson, Scott (2019c): InfluxQL functions, URL: https://docs.InfluxData.com/InfluxDB/v1.7/query_language/functions/, Stand: 29. Juli 2019.

Bang, Steve/Anderson, Scott (2019d): Installing InfluxDB OSS, URL: https://docs.InfluxData.com/InfluxDB/v1.7/introduction/installation/, Stand: 5. Mai 2019.

Bang, Steve/Schmitz, Johann (2019): Telegraf input plugins, URL: https://docs.InfluxData.com/Telegraf/v1.10/plugins/inputs/, Stand: 22. Mai 2019.

Betts, Dominic u. a. (2012): Exploring CQRS and Event Sourcing. A journey into high scalability, availability, and maintainability with Windows Azure.

Chandy, K. Mani (2009): Event Driven Architecture, in: Liu, Ling/Özsu, M. Tamer (Hrsg.): Encyclopedia of database systems, New York, NY, S. 1040–1044.

Chinchilla, Chris (2018a): Creating and writing to a stream, URL: https://eventstore.org/docs/http-api/creating-writing-a-stream/, Stand: 9. Juni 2019.

Chinchilla, Chris (2018b): Rolling Snapshots, URL: https://eventstore.org/docs/event-sourcing-basics/rolling-snapshots/index.html, Stand: 8. Mai 2019.

Chinchilla, Chris (2018c): User Defined Projections, URL: https://eventstore.org/docs/projections/user-defined-projections/index.html, Stand: 1. August 2019.

Chinchilla, Chris (2019a): Step 1 - Install, run, and write your first event, URL: https://eventstore.org/docs/getting-started/index.html?tabs=tabid-1%2Ctabid-dotnet-client%2Ctabid-dotnet-client-connect%2Ctabid-4, Stand: 7. Mai 2019.

Chinchilla, Chris (2019b): Step 2 - Read events from a stream and subscribe to changes, URL: https://eventstore.org/docs/getting-started/reading-subscribing-events/index.html?tabs=tabid-6%2Ctabid-dotnet-client%2Ctabid-8%2Ctabid-dotnet-read-event%2Ctabid-create-sub-http, Stand: 8. Mai 2019.

Chinchilla, Chris (2019c): Step 3 - Projections, URL: https://eventstore.org/docs/getting-started/projections/index.html?tabs=tabid-1%2Ctabid-4%2Ctabid-http-api%2Ctabid-create-proj-bash%2Ctabid-8%2Ctabid-update-proj-http%2Ctabid-reset-http%2Ctabid-read-stream-http%2Ctabid-update-proj-config-http%2Ctabid-read-projection-events-renamed-http%2Ctabid-enablebycategory-http%2Ctabid-projections-count-per-stream-http%2Ctabid-read-partition-http, Stand: 1. August 2019.

Cooper, Brian F. u. a. (2010): Benchmarking cloud serving systems with YCSB, in: Hellerstein, Joseph M./Chaudhuri, Surajit/Rosenblum, Mendel (Hrsg.): Proceed-

ings of the 1st ACM symposium on Cloud computing - SoCC '10, New York, New York, USA, S. 143–154.

Crowley, Noah (2018): Working with Irregular Time Series, URL: https://www.InfluxData.com/blog/working-with-irregular-time-series/, Stand: 1. Mai 2019.

DIN Deutsches Institut für Normung e. V. (2011): Ergonomie der Mensch-System-Interaktion – Teil 210: Prozess zur Gestaltung gebrauchstauglicher interaktiver Systeme, 9241-210, Berlin.

Dix, Paul (2016): Why Time-Series Matters For Metrics, Real-Time and Sensor Data. An InfluxData Whitepaper.

Druid (2019a): Community and Third Party Software, URL: http://Druid.io/libraries.html, Stand: 14. Januar 2019.

Druid (2019b): Frequently Asked Questions, URL: http://Druid.io/faq, Stand: 5. Januar 2019.

Dunkel, Jürgen u. a. (2008): Systemarchitekturen für Verteilte Anwendungen. Client-Server, Multi-Tier, SOA, Event-Driven Architectures, P2P, Grid, Web 2.0, München.

Dunning, Ted/Friedman, B. Ellen (2014): Time Series Databases. New Ways to Store and Access Data, Sebastopol, CA.

Ericson, AnnMarie/Berndtsson, Mikael/Mellin, Jonas (2009): Event in Active Databases, in: Liu, Ling/Özsu, M. Tamer (Hrsg.): Encyclopedia of database systems, New York, NY, S. 1044–1045.

Event Store LLP (2018a): Admin UI, URL: https://eventstore.org/docs/server/admin-ui/index.html#users, Stand: 15. Januar 2019.

Event Store LLP (2018b): Commercial Support, URL: https://eventstore.org/support/, Stand: 16. Januar 2019.

Event Store LLP (2018c): Event Sourcing Basics, URL: https://eventstore.org/docs/event-sourcing-basics/, Stand: 5. Dezember 2018.

Event Store LLP (2018d): Event Store as a functional database, URL: https://eventstore.org/docs/event-sourcing-basics/event-store-as-a-functional-database/index.html, Stand: 6. Januar 2019.

Event Store LLP (2018e): Performance and Scalability, URL: https://eventstore.org/docs/event-sourcing-basics/performance-and-scaling/index.html, Stand: 8. Januar 2019.

Event Store LLP (2018f): The open-source, functional database with Complex Event Processing in JavaScript., URL: https://eventstore.org/, Stand: 6. Januar 2019.

Fowler, Martin (2005): Event Sourcing, URL: https://martinfowler.com/eaaDev/EventSourcing.html, Stand: 9. Dezember 2018.

Freedman, Mike (2019): TimescaleDB 1.2: Analytical functions, automated data lifecycle management, improved performance, and more, URL: https://blog.timescale.com/TimescaleDB-1-2-analytical-functions-advanced-data-lifecycle-management-improved-performance/, Stand: 2. Mai 2019.

Gao, Like./Wang, Xiaoyang Sean (2005): Continuous Similarity-Based Queries on Streaming Time Series, in: IEEE Transactions on Knowledge and Data Engineering, 17. Jg., Nr. 10, S. 1320–1332.

Giorgetti, Alessandro (2019a): Architectural Overview, URL: https://github.com/NEventStore/NEventStore/wiki/Architectural-Overview, Stand: 7. Mai 2019.

Giorgetti, Alessandro (2019b): Home, URL: https://github.com/NEventStore/NEventStore/wiki, Stand: 9. Mai 2019.

Giorgetti, Alessandro (2019c): NEventStore, URL: https://github.com/NEventStore/NEventStore, Stand: 7. Mai 2019.

Giorgetti, Alessandro (2019d): Supported Persistence Engines, URL: https://github.com/NEventStore/NEventStore/wiki/Supported-Persistence-Engines, Stand: 7. Mai 2019.

Grafana Labs (2018): Plugins, URL: https://grafana.com/plugins, Stand: 14. Januar 2019.

Grimm, Rainer (2009): Funktionale Programmierung (1): Grundzüge, URL: http://www.linux-magazin.de/ausgaben/2009/09/funktionale-programmierung-1-grundzuege/5/, Stand: 6. Januar 2019.

Härder, Theo/Rahm, Erhard (2001): Datenbanksysteme. Konzepte und Techniken der Implementierung, Berlin, Heidelberg.

HL7 Deutschland e.V. (o.J.): HL7 CDA – Clinical Document Architecture, URL: http://hl7.de/themen/hl7-cda-clinical-document-architecture/, Stand: 24. April 2019.

Hong, Mingsheng/Demers, Alan/Gehrke, Johannes (2009): Event and Pattern Detection over Streams, in: Liu, Ling/Özsu, M. Tamer (Hrsg.): Encyclopedia of database systems, New York, NY, S. 1029–1033.

IBM (o.J.a): Developer guide for IBM Db2 Event Store client APIs, URL: https://www.ibm.com/support/knowledgecenter/en/SSGNPV_1.1.2/eventstore/develop/dev-guide.html#queryData, Stand: 8. Mai 2019.

IBM (o.J.b): IBM Db2 Event Store, URL: https://www.ibm.com/products/db2-event-store, Stand: 5. Mai 2019.

IBM (o.J.c): IBM Db2 Event Store Online Analytic Processing API guide, URL: https://www.ibm.com/support/knowledgecenter/en/SSGNPV_1.1.2/eventstore/develop/dev-eventsession.html, Stand: 8. Mai 2019.

IBM (o.J.d): System requirements for IBM Db2 Event Store Developer Edition, URL: https://www.ibm.com/support/knowledgecenter/SSGNPV_1.1.2/eventstore/desktop/requirements-desktop.html, Stand: 7. Mai 2019.

IBM (o.J.e): Welcome to IBM Db2 Event Store, URL: https://www.ibm.com/support/knowledgecenter/SSGNPV_1.1.3/welcome.html, Stand: 5. Mai 2019.

IBM (o.J.f): Welcome to IBM Db2 Event Store Developer Edition, URL: https://www.ibm.com/support/knowledgecenter/SSGNPV_1.1.3/desktop/welcome.html#welcome, Stand: 9. Mai 2019.

InfluxData, Inc. (2019a): Chronograf, URL: https://www.InfluxData.com/time-series-platform/Chronograf/, Stand: 1. Juni 2019.

InfluxData, Inc. (2019b): Compare, URL: https://www.InfluxData.com/products/compare/, Stand: 15. Januar 2019.

InfluxData, Inc. (2019c): Hardware sizing guidelines, URL: https://docs.InfluxData.com/InfluxDB/v1.7/guides/hardware_sizing/, Stand: 5. Mai 2019.

InfluxData, Inc. (2019d): InfluxDB is the Time Series Database in the TICK Stack, URL: https://www.InfluxData.com/time-series-platform/InfluxDB/, Stand: 5. Januar 2019.

InfluxData, Inc. (2019e): InfluxDB-comparisons, URL: https://github.com/InfluxData/InfluxDB-comparisons, Stand: 14. Juli 2019.

InfluxData, Inc. (2019f): Services, URL: https://www.InfluxData.com/products/services/, Stand: 16. Januar 2019.

InfluxData, Inc. (2019g): Technical Papers, URL: https://www.InfluxData.com/_resources/techpapers-new/, Stand: 14. Juli 2019.

InfluxData, Inc. (2019h): Telegraf, URL: https://www.InfluxData.com/time-series-platform/Telegraf/, Stand: 22. Mai 2019.

KairosDB (2015): Rich features to answer your needs, URL: http://KairosDB.github.io/, Stand: 5. Januar 2019.

KairosDB Team (2015a): Getting Started, URL: http://KairosDB.github.io/docs/build/html/GettingStarted.html, Stand: 5. Mai 2019.

KairosDB Team (2015b): KairosDB documentation v1.2.0, URL: http://KairosDB.github.io/docs/build/html/index.html, Stand: 16. Januar 2019.

KairosDB Team (2015c): Query Metrics, URL: http://KairosDB.github.io/docs/build/html/restapi/QueryMetrics.html, Stand: 1. Mai 2019.

KairosDB Team (2015d): Querying data, URL: http://KairosDB.github.io/docs/build/html/QueryingData.html, Stand: 1. Mai 2019.

KairosDB Team (2015e): Roll-ups, URL: http://KairosDB.github.io/docs/build/html/Roll-ups.html, Stand: 2. Mai 2019.

Kholod, Ivan u. a. (2017): Time Series Distributed Analysis in IoT with ETL and Data Mining Technologies, in: Galinina, Olga u. a. (Hrsg.): Internet of Things, Smart Spaces, and Next Generation Networks and Systems. 17th International Conference, NEW2AN 2017, 10th Conference, ruSMART 2017, Third Workshop NsCC 2017, St. Petersburg, Russia, August 28–30, 2017, Proceedings, Cham, S. 97–108.

Kulkarni, Ajay (2017): What the heck is time-series data (and why do I need a time-series database)?, URL: https://blog.timescale.com/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563, Stand: 2. Dezember 2018.

Leech, Dan/Chinchilla, Chris/Knight, Rich (2019): Welcome to Event Store, URL: https://eventstore.org/docs/, Stand: 9. Mai 2019.

Litzel, Nico (2018): Was ist ein BLOB?, URL: https://www.bigdata-insider.de/was-ist-ein-blob-a-732626/, Stand: 10. Januar 2019.

Liu, Rui/Yuan, Jun (2019): Benchmarking Time Series Databases with IoTDB-Benchmark for IoT Scenarios.

Luber, Stefan/Litzel, Nico (2017a): Was ist eine In-Memory-Datenbank?, URL: https://www.bigdata-insider.de/was-ist-eine-in-memory-datenbank-a-655470/, Stand: 5. Mai 2019.

Luber, Stefan/Litzel, Nico (2017b): Was ist NoSQL?, URL: https://www.bigdata-insider.de/was-ist-nosql-a-615718/, Stand: 30. April 2019.

Luckham, David/Schulte, Roy (2011): Event Processing Glossary – Version 2.0, URL: http://www.complexevents.com/wp-content/uploads/2011/08/EPTS_Event_Processing_Glossary_v2.pdf, Stand: 1. Dezember 2018.

Merlino, Gian u. a. (2019): Retaining or Automatically Dropping Data, URL: http://Druid.io/docs/latest/operations/rule-configuration.html, Stand: 1. Mai 2019.

Microsoft (2019): What is .NET?, URL: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet, Stand: 7. Mai 2019.

Müller, Michael (2016): Enabling retroactive computing through event sourcing, Ulm.

OpenTSDB (2018a): Additional Resources, URL: http://OpenTSDB.net/docs/build/html/resources.html#clients, Stand: 14. Januar 2019.

OpenTSDB (2018b): Documentation for OpenTSDB 2.3, URL: http://OpenTSDB.net/docs/build/html/index.html, Stand: 16. Januar 2019.

OpenTSDB (2018c): HBase Schema, URL: http://OpenTSDB.net/docs/build/html/user_guide/backends/HBase.html#data-table-schema, Stand: 9. Januar 2019.

OpenTSDB (2019a): Installation, URL: http://OpenTSDB.net/docs/build/html/installation.html#runtime-requirements, Stand: 5. Mai 2019.

OpenTSDB (2019b): Querying or Reading Data, URL: http://OpenTSDB.net/docs/build/html/user_guide/query/index.html, Stand: 1. Mai 2019.

OpenTSDB (2019c): Rollup And Pre-Aggregates, URL: http://OpenTSDB.net/docs/build/html/user_guide/rollups.html, Stand: 2. Mai 2019.

Overeem, Michiel/Spoor, Marten/Jansen, Slinger (2017): The Dark Side of Event Sourcing: Managing Data Conversion, in: Pinzger, Martin/Bavota, Gabriele/Marcus, Andrian (Hrsg.): 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). February 21-24, 2017, Klagenfurt, Austria, Piscataway, NJ, S. 193–204.

Rothsberg, Johan (2015): Evaluation of using NoSQL databases in an event sourcing system, Linköping.

Rybicki, Jedrzej (2018): Application of Event Sourcing in Research Data Management, in: Weckman, Gary/Grzymala-Busse, Jerzy (Hrsg.): ALLDATA 2018. The Fourth International Conference on Big Data, Small Data, Linked Data and Open Data : April 22-26, 2018, Athen, S. 46–52.

Seidemann, Marc/Seeger, Bernhard (2017): ChronicleDB: A High-Performance Event Store, in: Markl, Volker u. a. (Hrsg.): Advances in database technology - EDBT 2017. 20th International Conference on Extending Database Technology, Venice, Italy, March 21-24, 2017 : proceedings, Konstanz, S. 144–155.

Shoshani, Arie (2009): Temporal Logical Models, in: Liu, Ling/Özsu, M. Tamer (Hrsg.): Encyclopedia of database systems, New York, NY, S. 2992–2998.

solid IT gmbh (2019): DB-Engines Ranking von Event Stores, URL: https://db-engines.com/de/ranking/event+store, Stand: 5. Mai 2019.

Taylor, Hugh u. a. (2009): Event-Driven Architecture. How SOA Enables the Real-Time Enterprise, Upper Saddle River, NJ.

The OpenTSDB Authors (2010a-2019): How does OpenTSDB work?, URL: http://OpenTSDB.net/overview.html, Stand: 5. Januar 2019.

The OpenTSDB Authors (2010b-2019): The Scalable Time Series Database, URL: http://OpenTSDB.net/, Stand: 5. Januar 2019.

The PostgreSQL Global Development Group (1996a-2019): About, URL: https://www.PostgreSQL.org/about/, Stand: 1. Mai 2019.

The PostgreSQL Global Development Group (1996b-2019): Linux downloads (Ubuntu), URL: https://www.PostgreSQL.org/download/linux/ubuntu/, Stand: 5. Mai 2019.

The PostgreSQL Global Development Group (1996c-2019): PostgreSQL 11.2 Documentation, URL: https://www.PostgreSQL.org/files/documentation/pdf/11/PostgreSQL-11-A4.pdf.

The PostgreSQL Global Development Group (1996d-2019): Support, URL: https://www.PostgreSQL.org/support/, Stand: 16. Januar 2019.

Timescale, Inc. (2018a): Data Retention, URL: https://docs.timescale.com/v1.2/using-TimescaleDB/data-retention, Stand: 2. Mai 2019.

Timescale, Inc. (2018b): FAQ, URL: https://docs.timescale.com/v1.2/faq, Stand: 2. Mai 2019.

Timescale, Inc. (2018c): Installation, URL: https://docs.timescale.com/v1.2/getting-started/installation/ubuntu/installation-apt-ubuntu, Stand: 5. Mai 2019.

Timescale, Inc. (2018d): TimescaleDB Documentation, URL: https://docs.timescale.com/v1.1/main, Stand: 5. Mai 2019.

Timescale, Inc. (2019a): Open-source time-series database powered by PostgreSQL, URL: https://www.timescale.com/, Stand: 5. Januar 2019.

Timescale, Inc. (2019b): Visualizing data, URL: https://docs.timescale.com/v1.1/using-TimescaleDB/visualizing-data, Stand: 14. Januar 2019.

Wei, Jonathan u. a. (2019a): Druid Quickstart, URL: http://Druid.io/docs/latest/tutorials/index.html, Stand: 5. Mai 2019.

Wei, Jonathan u. a. (2019b): Querying, URL: http://Druid.io/docs/latest/querying/querying.html, Stand: 1. Mai 2019.

Wei, Jonathan u. a. (2019): SQL, URL: http://Druid.io/docs/latest/querying/sql.html, Stand: 1. Mai 2019.

Wei, Jonathan u. a. (2019c): What is Druid?, URL: http://Druid.io/docs/latest/design/, Stand: 16. Januar 2019.

Wei, Jonathan/Lim, David (2019): Tutorial: Roll-up, URL: http://Druid.io/docs/latest/tutorials/tutorial-rollup.html, Stand: 1. Mai 2019.

Ye, Brian (2017): An Evaluation on Using Coarse-grained Events in an Event Sourcing Context and its Effects Compared to Fine-grained Events, Stockholm.

Young, Greg (2010a): CQRS and Event Sourcing, URL: http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/.

Young, Greg (2010b): CQRS Documents by Greg Young, URL: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf, Stand: 4. Dezember 2018.

Anhang

Anhang A: R-Skript „basic-dataPreparation.R”

```
## Definition of function for basic data preparation and anonymization of Apple Health Data

folder <- # path to folder
file <-  # name of file

dataPreparation <- function(folder, file) {
  user <- dirname(dirname(file)) %>% gsub("Export", "User", .)

  xml <- xmlParse(glue(folder, file))

  # transform xml file to data frame - select the Record rows from the xml file

  df <- XML:::xmlAttrsToDataFrame(xml["//Record"])

  # exclude data with no device / unit

  df <- df %>%
    filter(! is.na(device)) %>%
    filter(! is.na(unit))

  # change data type of value, device, sourceName, unit

  df <- df %>%
    mutate(value = as.integer(as.character(value))) %>%
    mutate(device = as.character(device)) %>%
    mutate(sourceName = as.character(sourceName)) %>%
    mutate(unit = as.character(unit))

  # change data type of Dates in a date time variable POSIXct (Central European Time) using lubridate

  df <- df %>%
    mutate(endDate = with_tz(ymd_hms(endDate), tzone = "CET")) %>%
    mutate(startDate = with_tz(ymd_hms(startDate), tzone = "CET")) %>%
    mutate(creationDate = with_tz(ymd_hms(creationDate), tzone ="CET"))


  # add a new column to df with measurement

  df <- df %>%
    mutate(measurement =
             ifelse(starts_with("HKQuantityTypeIdentifier",ignore.case = TRUE, vars = type ),substr(type, 25, 1000),0))

   # add a new column to df with user ID

  df <- df %>%
    mutate(userID = user)

  # add a new column to df with anonymous device

  df <- df %>%
    mutate(deviceAnonymous = sapply(strsplit(device, "name:"), "[", 2)) %>%
    mutate(deviceAnonymous = sapply(strsplit(deviceAnonymous, ", manufacturer:"), "[", 1))

  # add new columns to df with userDeviceID & userDeviceSourceID

  df <- df %>%
    mutate(userDeviceID = paste(userID, deviceAnonymous, sep = "_")) %>%
    mutate(userDeviceSourceID = paste(userDeviceID, group_indices(., sourceName), sep = "_"))

   # select relevant columns for database insertion

  df <- df %>%
    select(unit, startDate, endDate, value, userID, deviceAnonymous, userDeviceSourceID, measurement)

}
```

Anhang B: R-Skript „InfluxDB-dataPreparation.R”

```
## definition of function for basic data preparation and anonymization of Apple Health Data

InfluxDBdataPreparation <- function(folder, file) {
  user <- dirname(dirname(file)) %>% gsub("Export", "User", .)

  ########################################################
  # basic data preparation
  ########################################################

 xml <- xmlParse(glue(folder, file))

  # transform xml file to data frame - select the Record rows from the xml file

  df <- XML:::xmlAttrsToDataFrame(xml["//Record"])

  # exclude data with no device / unit

  df <- df %>%
    filter(! is.na(device)) %>%
    filter(! is.na(unit))

  # change data type of value, device, sourceName, unit

  df <- df %>%
    mutate(value = as.integer(as.character(value))) %>%
    mutate(device = as.character(device)) %>%
    mutate(sourceName = as.character(sourceName)) %>%
    mutate(unit = as.character(unit))

  # change data type of Dates in a date time variable POSIXct (Central European Time) using lubridate

  df <- df %>%
    mutate(endDate = with_tz(ymd_hms(endDate), tzone = "CET")) %>%
    mutate(startDate = with_tz(ymd_hms(startDate), tzone = "CET")) %>%
    mutate(creationDate = with_tz(ymd_hms(creationDate), tzone ="CET"))
  # add a new column to df with measurement

  df <- df %>%
    mutate(measurement =
             ifelse(starts_with("HKQuantityTypeIdentifier",ignore.case = TRUE, vars = type ),substr(type, 25, 1000),0))

   # add a new column to df with user ID

  df <- df %>%
    mutate(userID = user)

  # add a new column to df with anonymous device

  df <- df %>%
    mutate(deviceAnonymous = sapply(strsplit(device, "name:"), "[", 2)) %>%
    mutate(deviceAnonymous = sapply(strsplit(deviceAnonymous, ", manufacturer:"), "[", 1))

  # add new columns to df with userDeviceID & userDeviceSourceID

  df <- df %>%
    mutate(userDeviceID = paste(userID, deviceAnonymous, sep = "_")) %>%
    mutate(userDeviceSourceID = paste(userDeviceID, group_indices(., sourceName), sep = "_"))

  # InfluxDB - add a new column to df with time in Go reference time data type

  df <- df  %>%
    mutate(time = with_tz(endDate, tzone = "Europe/Berlin")) %>%
    mutate(time = format(time, format = "%a %b %d %H:%M:%S %z %Z %Y"))

  # InfluxDB - change data type startDate

  df <- df  %>%
    mutate(startDate = with_tz(startDate, tzone = "Europe/Berlin")) %>%
    mutate(startDate = format(startDate, format = "%a %b %d %H:%M:%S %z %Z %Y")) %>%

  # select relevant columns for database insertion

  df <- df %>%
    select(unit, startDate, time, value, userID, deviceAnonymous, userDeviceSourceID, measurement)

  ########################################################
  # preparation for CSV
  ########################################################

  # InfluxDB - save as csv file

  write_csv(df, file.path(paste0(folder.cleanCSV, user, ".csv")))

}
```

Anhang C: R-Skript „InfluxDB-loop-dataPreparation.R”

```
library(tidyverse)
library(lubridate)
library(XML)
library(glue)
library(pbapply)

Sys.setlocale("LC_ALL","English")

source("InfluxDB-dataPreparation.R")

# load apple health export.xml files

folder <- # path to folder
folder.cleanCSV <- glue(dirname(folder), "/cleanCSV/")


files <- list.files(folder, "Export.xml", recursive = T)

# apply function InfluxDBdataPreparation on all apple health export.xml files

pbsapply(files, function(file) {
  InfluxDBdataPreparation(folder, file)
  })
```

Anhang D: R-Skript „EventStore-loop-dataPreparation.R“

```
library(tidyverse)
library(lubridate)
library(XML)
library(glue)
library(pbapply)
library(RJSONIO)
library(uuid)

source("EventStore-dataPreparation.R")
source("EventStore-SingleEventPreparation.R")

# load apple health export.xml files

folder <- # path to folder

files <- list.files(folder, "Export.xml", recursive = T)

# definitin of batchSize = no of events in one JSON
batchSize <- 60000

# create a class definition for "data" in event JSON

setClass("event.data", representation(value ="numeric",
                                      startDate ="character",
                                      endDate ="character",
                                      deviceAnonymous ="character",
                                      userDeviceSourceID ="character"))

# create a class definition for "metadata" in event JSON

setClass("event.metadata", representation(unit ="character"))

# create a class definition for the event JSON

setClass("event", representation(eventId = "character",
                                 eventType = "character",
                                 data = "event.data",
                                 metadata ="event.metadata"))

# apply function InfluxDBdataPreparation on all apple health export.xml files

pbsapply(files, function(file) {
  EventStoredataPreparation(folder, file)
  })
```

Anhang E: R-Skript „EventStore-dataPreparation.R“

```r
## definition of function for event store data preparation and anonymization of Apple Health Data

EventStoredataPreparation <- function(folder, file) {

  user <- dirname(dirname(file)) %>% gsub("Export", "User", .)

  ########################################################
  # basic data preparation
  ########################################################

  xml <- xmlParse(glue(folder, file))

  #transform xml file to data frame - select the Record rows from the xml file

  df <- XML:::xmlAttrsToDataFrame(xml["//Record"])

  # exclude data with no device / unit

  df <- df %>%
    filter(! is.na(device)) %>%
    filter(! is.na(unit))

  # change data type of value, device, sourceName, unit

  df <- df %>%
    mutate(value = as.integer(as.character(value))) %>%
    mutate(device = as.character(device)) %>%
    mutate(sourceName = as.character(sourceName)) %>%
    mutate(unit = as.character(unit))

  # change data type of Dates in a date time variable POSIXct using lubridate, then as character

  df <- df %>%
    mutate(endDate = as.character(with_tz(ymd_hms(endDate), tzone = "CET"))) %>%
    mutate(startDate = as.character(with_tz(ymd_hms(startDate), tzone ="CET"))) %>%
    mutate(creationDate = as.character(with_tz(ymd_hms(creationDate), tzone = "CET")))

  # add a new column to df with measurement

  df <- df %>%
    mutate(measurement =
             ifelse(starts_with("HKQuantityTypeIdentifier",ignore.case = TRUE, vars = type ),substr(type, 25, 1000),0))

  # add a new column to df with user ID

  df <- df %>%
    mutate(userID = user)

  # add a new column to df with anonymous device

  df <- df %>%
    mutate(deviceAnonymous = sapply(strsplit(device, "name:"), "[", 2)) %>%
    mutate(deviceAnonymous = sapply(strsplit(deviceAnonymous, ", manufacturer:"), "[", 1))

  # add new columns to df with userDeviceID & userDeviceSourceID

  df <- df %>%
    mutate(userDeviceID = paste(userID, deviceAnonymous, sep = "_")) %>%
    mutate(userDeviceSourceID = paste(userDeviceID, group_indices(., sourceName), sep = "_"))

  # select relevant columns for database insertion

  df <- df %>%
    select(unit, startDate, endDate, value, userID, deviceAnonymous, userDeviceSourceID, measurement)

  ########################################################
  # preparation for JSON
  ########################################################

  # counts number of rows of one df = export of one user
  numberOfRows <- nrow(df)

  # calculates the number of batches/JSONs needed to parse all data from dataframe
  numberOfBatches <- (numberOfRows %/% batchSize)+1

  #
  if (numberOfBatches > 1) {

    # defines first & last row of data from for each batch

    firstRow <- 1
    lastRow <- batchSize

    ## loops over needed number of batches -1
    for (batchNumber in 1:(numberOfBatches-1)) {
      vector <- character(batchSize)

      newdf <- df[firstRow:lastRow,]

      # loops over every row in a batch

      for (rownumber in 1:batchSize) {
        singleRow <- SingleEventPreparation(newdf, rownumber)
        json <- toJSON(singleRow)
        vector[rownumber] <- json
      }
```

```
    # pastes single JSON objects to required format

    jsonObjects <- paste(vector, collapse = ",")

    finalJson <- paste("[", jsonObjects, "]")

    # file & path to save JSON files to

    outputFile <- paste0(user,"-batch-",batchNumber,".json")
    outputPath <- paste0(" # path to output folder", outputFile)

    cat(finalJson, file = outputPath)

    firstRow <- firstRow + batchSize
    lastRow <- lastRow + batchSize
  }

    ## last batch (number of remaining rows < batchSize )

    # counts the number of rows of the last batch

    lastBatchSize <- numberOfRows %% batchSize

    vector <- character(lastBatchSize)

    newdf <- df[firstRow:numberOfRows,]

    # loops over every row in a batch

    for (rownumber in 1:lastBatchSize) {
      singleRow <- SingleEventPreparation(newdf, rownumber)
      json <- toJSON(singleRow)
      vector[rownumber] <- json
    }

    # pastes single JSON objects to required format

    jsonObjects <- paste(vector, collapse = ",")

    finalJson <- paste("[", jsonObjects, "]")

    # file & path to save JSON files to

    outputFile <- paste0(user,"-batch-",numberOfBatches,".json")
    outputPath <- paste0(" # path to output folder", outputFile)

    cat(finalJson, file = outputPath)

} else {

    ## single batch (number of rows in df < batchSize )

    vector <- character(numberOfRows)

    newdf <- df[1:numberOfRows,]

    # loops over every row in a batch

    for (rownumber in 1:numberOfRows) {
      singleRow <- SingleEventPreparation(newdf, rownumber)
      json <- toJSON(singleRow)
      vector[rownumber] <- json
    }

    # pastes single JSON objects to required format

    jsonObjects <- paste(vector, collapse = ",")

    finalJson <- paste("[", jsonObjects, "]")

    # file & path to save JSON files to

    outputFile <- paste0(user,"-batch-",numberOfBatches,".json")
    outputPath <- paste0(" # path to output folder", outputFile)

    cat(finalJson, file = outputPath)

}

 }
```

Anhang F: R-Skript „EventStore-SingleEventPreparation.R“

```
## definition of function to convert one row of the df to JSON object

SingleEventPreparation <- function(inputDataframe, rowIndex) {
  event.metadata <- new("event.metadata",
                        unit = inputDataframe[rowIndex,"unit"])
  event.data <- new("event.data",
                    value = inputDataframe[rowIndex,"value"],
                    startDate = inputDataframe[rowIndex,"startDate"],
                    endDate = inputDataframe[rowIndex,"endDate"],
                    deviceAnonymous = inputDataframe[rowIndex,"deviceAnonymous"],
                    userDeviceSourceID = inputDataframe[rowIndex,"userDeviceSourceID"])
  event <- new("event", eventId= UUIDgenerate(),
               eventType = inputDataframe[rowIndex,"measurement"],
               data = event.data ,
               metadata = event.metadata)
  event
}
```

Anhang G: Anzahl der notwendigen Batches je Batch Größe

		# notwenige Batches bei unterschiedliechen Batch Größen							
Export Nr	Messungen	10.000	20.000	30.000	40.000	50.000	60.000	70.000	80.000
Export1	48.826	5	3	2	2	1	1	1	1
Export2	2.697	1	1	1	1	1	1	1	1
Export3	1.830	1	1	1	1	1	1	1	1
Export4	58.013	6	3	2	2	2	1	1	1
Export5	40.469	5	3	2	2	1	1	1	1
Export6	4.768	1	1	1	1	1	1	1	1
Export7	81.531	9	5	3	3	2	2	2	2
Export8	78.000	8	4	3	2	2	2	2	1
Export9	78.498	8	4	3	2	2	2	2	1
Export10	32.806	4	2	2	1	1	1	1	1
Export11	553.493	56	28	19	14	12	10	8	7
Export12	165.134	17	9	6	5	4	3	3	3
Export13	253.103	26	13	9	7	6	5	4	4
Export14	47.064	5	3	2	2	1	1	1	1
Export15	186.767	19	10	7	5	4	4	3	3
Export16	351.251	36	18	12	9	8	6	6	5
Export17	303.465	31	16	11	8	7	6	5	4
Export18	50.446	6	3	2	2	2	1	1	1
Export19	221.990	23	12	8	6	5	4	4	3
Export20	57.477	6	3	2	2	2	1	1	1
Export21	39.470	4	2	2	1	1	1	1	1
Export22	96.531	10	5	4	3	2	2	2	2
Export23	39.174	4	2	2	1	1	1	1	1
Export24	32.568	4	2	2	1	1	1	1	1
Export25	4.829	1	1	1	1	1	1	1	1
Export26	393.863	40	20	14	10	8	7	6	5
Export27	40.320	5	3	2	2	1	1	1	1
	3.264.383	341	177	125	96	80	68	62	55

Batch Größe	10.000	20.000	30.000	40.000	50.000	60.000	70.000	80.000
Anzahl notwendiger Batches	341	177	125	96	80	68	62	55

Anhang H: R-Skript „EventStore-POST-batchWrite.R“

```
folder <- # path to folder

json.files <- list.files(folder)

vector <- vector(length = length(json.files))

for (file in 1:length(json.files)) {
  c <- unlist(strsplit(json.files[file], "-", 1))
  event.stream <- c[1]

  command <- paste0("curl -i -d \"@", json.files[file], "\" \"http://139.6.56.162:2113/streams/",
                    event.stream, "thesisKS50k\" -H \"Content-Type:application/vnd.eventstore.events+json\"")

  vector[file] <- command

}

write(vector, file = "# path to folder\batchwrite.sh")
```

TH Köln
Gustav-Heinemann-Ufer 54
50968 Köln
www.th-koeln.de

Technology
Arts Sciences
TH Köln